Université Sidi Mohammed Ben Abdellah
Ecole Nationale des Sciences Appliquées – Fès

STRUCTURES DE DONNÉES

Pr. Karima AISSAOUI
Karima.aissaoui1@usmba.ac.ma

2019/2020

Scanned with CamScanner

Plan du cours

- Rappel du langage C
- Listes chainées
- Listes doubles chainées
- Pile/file
- Arbres
- Tables de hachage
- Algorithme de tri

Scanned with CamScanner

Plan de la séance

- Les déclarations de variables
  - Types prédéfinis
  - Tableaux
- Instructions itératives
- Instructions conditionnelles
- Opérateurs de comparaison et d'assignation
- Allocation dynamique de mémoire, pointeurs
- Passages de variables, tableaux et fonctions
- Variables, tableaux et fonctions
- Spécification de type : const
- Fonctions

Scanned with CamScanner

Les déclarations de variables

- Le langage C est un langage fortement typé
  - Types prédéfinis

Type	Taille	borne inférieure	borne supérieure
short int	entier 2 o	−32768	+32767
int	entier 4 o	−2147483648	2147483647
long long	entier 8 o	−9223372036854775808	9223372036854775807
float	réel 4 o		

- Les nombres de type **float** sont codés sur 32 bits dont :
  - 23 bits pour la mantisse
  - 8 bits pour l'exposant
  - 1 bit pour le signe

double	réel 8 o	

- Les nombres de type **double** sont codés sur 64 bits dont :
  - 52 bits pour la mantisse
  - 11 bits pour l'exposant
  - 1 bit pour le signe

Le langage C respecte la casse (Majuscule est différent de minuscule).

Scanned with CamScanner

Les déclarations de variables

- En C, les tableaux commencent par défaut avec un indice égal à 0. La déclaration d'un tableau de 100 éléments de réels à double précision est

  ```
  double tab[100];
  ```

- Le premier élément est tab[0] et le dernier tab[99].

- Noter qu'en C, les crochets sont réservés aux tableaux et que les parenthèses sont réservées aux arguments de fonctions.

- Les variables déclarées dans l'en-tête du programme sont des variables globales.

Scanned with CamScanner

Les tableaux

- L'initialisation d'un tableau peut être faite dans la déclaration, par exemple

```
double taba[3]={0,1,2};
double tab[10]={1,1,1};
/* les 7 autres éléments sont mis à 0 */
double tabc[]={10,7,4};
```

Dans la dernière ligne, la dimension a été omise ; cela signifie qu'un tableau de trois éléments est créé.

- Les tableaux à deux dimensions se déclarent de la manière suivante

```
double taba[2][3]={{0,1,2},{3,4,0}};
```

qui correspond à un tableau de 2 lignes et 3 colonnes. Comme les éléments du tableau sont rangés par ligne.

  - on peut aussi initialiser le tableau de la manière suivante

```
double taba[2][3]={0,1,2,3,4,0} ;
```

Scanned with CamScanner

Les instructions itératives

- Si le nombre d'itérations est connu à l'avance, on utilise l'instruction for qui contient un indice de boucle (variable entière).

```
int i;
for (i=m;i<n;i+=step){…}
```

  - où step correspond au pas d'incrémentation.

- Avec une écriture type C++

```
for (int i=m;i<n;i+=step){…}
```

  Si n est inférieur ou égal à m, aucune instruction de la boucle ne sera exécutée.

- Si la valeur de step est égale à 1, on peut écrire une boucle plus simplement

```
for(int i=m;i<n;i++){….}
```

Scanned with CamScanner

Les instructions itératives

- Si le nombre d'itérations dépend d'une condition, il y a deux possibilités

  while(condition) { ... }

  les instructions de la boucle ne sont exécutées que si la condition est vérifiée.

  ```
  do
  {
  ...
  } while(condition);
  ```

Les instructions sont exécutées une première fois et la condition est alors testée.

- Attention : Les boucles while ou do-while ne sont interrompues que si la condition devient fausse en un nombre fini d'itérations. Parmi les erreurs classiques de programmation, il arrive qu'un programme n'arrive pas à se finir car la condition reste indéfiniment vraie

Scanned with CamScanner

if, case, switch, break

- Si la condition à examiner est simple ou double, l'instruction if est recommandée,

```
if (condition)
{ instructions1}
else
{ instructions2}
```

- Cette instruction signifie que si la condition est vraie les instructions du groupe 1 sont exécutées. Si la condition est fausse les instructions du groupe 2 sont exécutées.
- Dans le cas où il y a deux conditions imbriquées

```
if (condition1)
{if (condition2)
{ instructions1}
else
{ instructions2}
}
```

- Si les conditions 1 et 2 sont vraies, alors on exécute les instructions 1 ; si la condition 1 est vraie et 2 est fausse, alors on exécute les instructions 2 ; Si la condition 1 est fausse, quelle que soit la condition 2, alors on n'exécute rien.

Scanned with CamScanner

if, case, switch, break

- On comprend facilement que l'imbrication des instructions if-else peut devenir rapidement fastidieuse et donc source d'erreurs. Il convient donc d'être prudent dans la rédaction de ces lignes de code.
- Situation de branchements multiples. La condition à évaluer peut donner différentes valeurs entières. Les instructions switch, case et break ont la structure suivante

```
switch (valeur_entière)
{
case 0: instructions1;break;
case 4: instructions2;break;
case 6: instructions3;break;
default: instructionsN;break;
}
```

- Selon la valeur de la variable valeur_entière, on exécute des instructions différentes : Si la valeur est 0 on exécute les instructions 0, si la valeur est 4, les instructions 2, si la valeur est 6 les instructions 3, pour toute autre valeur les instructions N. Il est préférable de placer le cas le plus fréquent en premier dans la suite des instructions case, et ainsi de suite.

Scanned with CamScanner

Comparaison et assignation

- Pour formuler une expression logique, il est souvent nécessaire d'avoir un opérateur de comparaison, le langage C fournit plusieurs opérateurs :

  < inférieur(strictement) à

  > supérieur (strictement) à

  <= inférieur ou égal à

  >= supérieur ou égal à

  ! = différent de == égal à

- Il faut faire attention à l'opérateur de comparaison == qui est différent de l'opérateur d'assignation =.

Scanned with CamScanner

Comparaison et assignation

- Par exemple

  if (i==1){...} else {...}

  signifie que si i est égal à 1 le premier groupe d'instructions est exécuté

  - tandis que le groupe d'instructions suivant

  if (i=1){...} else {...}

  signifie qu'on affecte 1 à i, cette opération étant valide, i = 1 sera considéré comme une opération qui sera toujours vraie et donc le premier groupe d'instructions sera toujours exécuté

Scanned with CamScanner

Comparaison et assignation

- Il existe aussi des opérateurs de comparaison logiques
  - && et logique
  - || ou logique

- Outre l'opérateur = qui est l'opérateur d'assignation déjà vu, il existe en C plusieurs autres opérateurs

  ++ incrémente d'une unité

  −− décrémente d'une unité

  + = ajoute le membre de droite au membre de gauche

  − = soustrait le membre de droite au membre de gauche

  ∗ = multiplie le membre de droite au membre de gauche

  / = divise le membre de droite au membre de gauche

Scanned with CamScanner

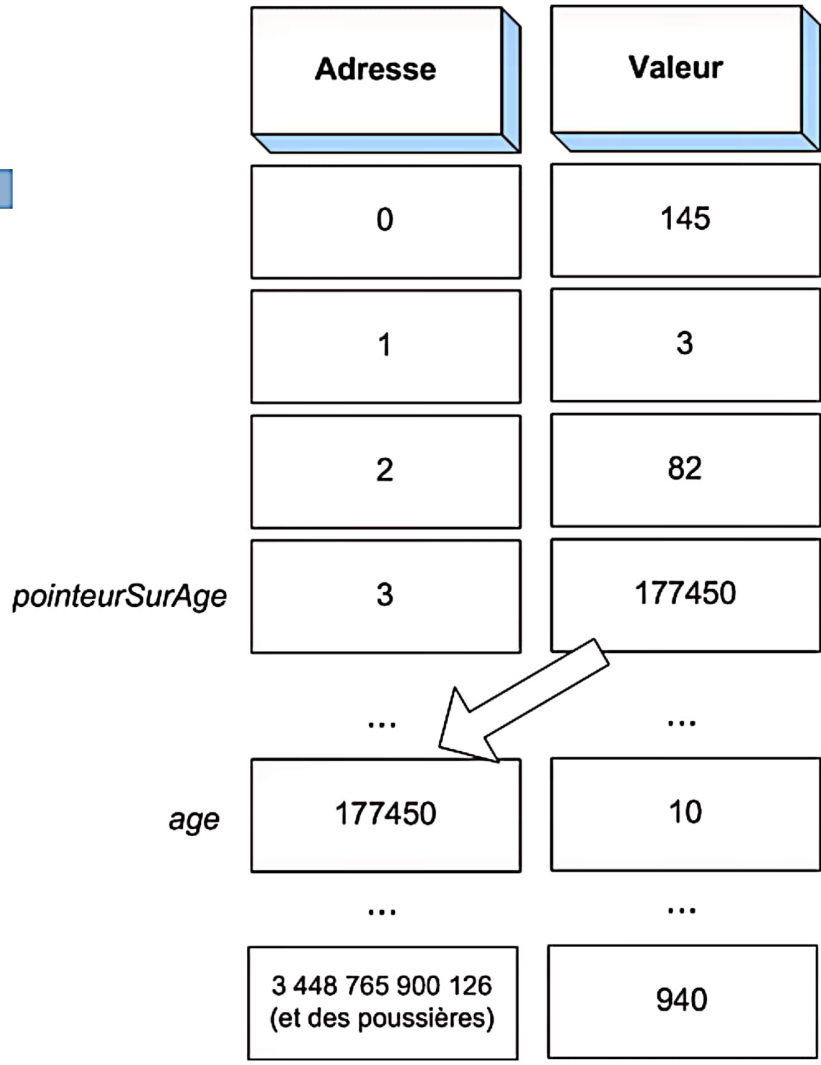
Alloc. dynamique, pointeurs

- Le langage C permet de modifier l'espace mémoire disponible pour le programme, à condition bien entendu que la machine puisse en fournir.
- Cette fonctionnalité très intéressante a une contrepartie qui est l'utilisation de pointeurs.
- Une fois une variable déclarée et initialisée, par exemple une variable a, on peut avoir accès à l'adresse de cette variable par l'opérateur d'indirection &.
- Par exemple &a donne la valeur de l'adresse de a. L'opération inverse qui consiste à avoir la valeur de la case mémoire possédant l'adresse stockée par la variable b, est effectuée par l'opérateur *.
- La variable b doit être alors ce que l'on appelle un pointeur.

Scanned with CamScanner

Pointeurs

```
int age = 10;
int *pointeurSurAge;
pointeurSurAge = &age;
```



Scanned with CamScanner

Allocation dynamique

- Un pointeur est une variable qui contient l'adresse d'une variable d'un type donné. La déclaration d'un pointeur se fait de la manière suivante ; type du pointeur, opérateur * et le nom du pointeur

```
double *b;
```

- On peut initialiser un pointeur en lui fournissant l'adresse d'une variable

```
b=&a;
```

- Si on veut avoir accès à la valeur pointée par b, on réalise l'opération

```
double c;
c=*b;
```

- La valeur de c est bien entendu celle de a

Scanned with CamScanner

Allocation dynamique

- Dans le cas d'un tableau, les éléments sont disposés séquentiellement en mémoire.

- Ainsi le nom d'un tableau donne l'adresse du premier élément du tableau.

Par exemple, pour un tableau initialement déclaré comme

```
double a[10];
```

- a (ou de manière moins élégante &a[0]) donne l'adresse .

- L'allocation dynamique de mémoire se fait de la manière suivante : on déclare un pointeur avec le type correspondant au tableau que l'on désire créer.

```
double * a=NULL;
```

Scanned with CamScanner

Allocation dynamique

- En initialisant le pointeur à l'adresse NULL, on évite de récupérer une adresse incorrecte et de créer des bugs.
- on utilise la fonction système malloc (ou calloc) dont la déclaration a été faite dans le fichier d'en-tête stdlib.h.
- La syntaxe est la suivante

  a= (double *) malloc(sizeof(double)*10);
- si la machine ne dispose pas de la mémoire demandée en cours d'exécution du programme, l'allocation ne se fait pas.
- Une manière simple de le savoir est de vérifier une fois l'allocation dynamique effectuée que le pointeur a possède la valeur d'une adresse différente de la valeur assignée au départ.

Scanned with CamScanner

Allocation dynamique

- Par exemple, l'instruction suivante placée après l'instruction malloc conduit à une interruption du programme si l'allocation dynamique n'a pas fonctionné, ce qui évite un arrêt dans l'exécution à un endroit du programme où l'on fait appel à un tableau qui n'a pas pu être créé.

```
if (a == NULL)
{fprintf(stderr," l'allocation dynamique n'a pas fonctionnée \n");
return(EXIT_FAILURE);}
```

- Une fois crée, les éléments du tableau sont accessibles à partir d'une syntaxe tout à fait identique à celle des tableaux statiques.

- Avant de quitter le programme il est nécessaire de libérer la mémoire qui a été demandée par l'utilisateur. L'instruction correspondante est alors

```
free(a);
```

Scanned with CamScanner

Allocation dynamique

- Attention au fait qu'il ne faut libérer la mémoire réservée pour le tableau a qu'une seule fois.
- Si l'instruction est exécutée deux fois, le programme risque de se planter.
- Inversement, si on oublie de libérer la mémoire, on risque de provoquer une fuite de mémoire ("memory leak").
- En résumé
  - Déclaration d'un pointeur (avec son type)
  - Appel de la fonction malloc
  - Vérification que la mémoire est bien allouée
  - Libération de la mémoire à la fin du programme.

Scanned with CamScanner

Allocation dynamique

- Il est possible de définir des types plus élaborés et aussi de définir des types à l'aide de l'instruction typedef.

- Par exemple, en plaçant l'instruction suivante dans le fichier d'en-tête du programme

```
typedef    long      int       entier_long;
```

- on peut par la suite définir des entiers longs de la manière suivante

```
entier_long    a,b,c;
```

Scanned with CamScanner

Allocation dynamique

- On peut aussi définir des types nouveaux à travers des structures. Par exemple

```
typedef struct {
long i;
double x,y;
} num_pos_points;
```

- Une fois ainsi défini en en-tête du programme, on peut utiliser ce type comme un type habituel.

- Soit la déclaration de la structure cc

```
num_pos_points cc;
```

- La syntaxe pour l'affectation des membres de la structure sera la suivante

```
cc.i=5;
cc.x=0.6;
cc.y=0.7;
```

Scanned with CamScanner

Allocation dynamique

- Pour un tableau de structure, on a la syntaxe de déclaration

```
num_pos_points cc[10];
```

- Pour le programme

```
for(int i=0;i<10;i++)
{ cc[i].i=5*i;
cc[i].x=0.6*i;
cc[i].y=0.7*i;
}
```

Scanned with CamScanner

Allocation dynamique

- Pour un tableau crée dynamiquement, la syntaxe du programme reste identique.

```
#include<stdlib.h>
#include<stdio.h>
typedef struct {
      int i;
      double x,y;
} num_pos_points;
main(){
      int j;
      num_pos_points *cc=NULL;
      cc= ( num_pos_points *) malloc(sizeof( num_pos_points)*10);
      if (cc == NULL)
      {
          fprintf(stderr," l'allocation dynamique n'a pas fonctionnée \n");
          return(EXIT_FAILURE);
      }
       for( j=0;j<10;j++)
        {
            cc[j].i=j;
            cc[j].x=0.6*j;
            cc[j].y=0.7*j;
        }
free(cc);
exit(EXIT_SUCCESS);}
```

Scanned with CamScanner

```
#include<stdlib.h>
#include<stdio.h>
typedef struct {
   int i;
   double x,y;
} num_pos_points;
main(){
   int j;
   num_pos_points *cc=NULL;
   cc= ( num_pos_points *) malloc(sizeof( num_pos_points)*10);
   if (cc == NULL)
   {
       fprintf(stderr," l'allocation dynamique n'a pas fonctionnée \n");
       return(EXIT_FAILURE);
     }
    for( j=0;j<10;j++)
     {
        cc[j].i=j;
        cc[j].x=0.6*j;
        cc[j].y=0.7*j;
       }
free(cc);
exit(EXIT_SUCCESS);}
```

Scanned with CamScanner

Variables, des tableaux et des fonctions

- Par construction, quand on écrit un programme, on a des objets qui sont généralement des nombres, des tableaux de nombres voire des tableaux de structure de nombres. Les fonctions que l'on crée permettent soit de modifier ces tableaux, soit de calculer des nouveaux nombres.

- Pour respecter une programmation structurée, les variables d'une fonction sont de trois types : il y a des variables d'entrée, des variables de sortie et des variables de travail (locales).

- Pour éviter la création de bugs, il est largement recommandé
  - que les variables d'entrée ne soient pas modifiées quand on exécute la fonction,
  - que les variables de travail de la fonction soient locales dans le sens où leur existence ne modifie pas les variables des autres fonctions et du programme appelant.
  - Une fois ces contraintes satisfaites, il ne reste qu'à s'occuper des variables de sortie qui, par nature, doivent modifier l'état des variables du programme appelant.

Scanned with CamScanner

Variables, tableaux et fonctions

- Nous allons maintenant examiner les différentes façons de passer des objets à une fonction.

- Pour ce qui est des variables d'entrée, la situation est simple car le langage C passe les variables par valeur, ce qui signifie qu'il recopie les valeurs transmises dans un espace de la mémoire propre à la fonction et que cette emplacement sera a priori libéré une fois cette fonction exécutée.

- Cette règle a l'avantage de ne pas modifier les variables d'entrée sans que l'on s'en aperçoive.

- Si l'on veut que la variable soit une variable de sortie, on a deux solutions : soit on donne un type à la fonction ce qui fait que la valeur sera récupérée dans le programme appelant, soit on transmet la valeur de l'adresse ce qui permet de modifier l'état de la variable initialement déclarée dans le programme appelant.

Scanned with CamScanner

Variables, tableaux et fonctions

- Pour les tableaux, on transmet toujours l'adresse d'un élément du tableau (généralement le premier) et le sous-programme peut modifier les valeurs du tableau transmis dans les arguments de la fonction.

Scanned with CamScanner

Variables, tableaux et fonctions

- En exécutant l'exemple ci-dessous, on peut vérifier que le tableau du programme principal a été modifié par la fonction inittab

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
void inittab(double *,int );
int main(){
        double x[10];
        inittab(x,10);
        for(int i=0;i<10;i++)
                {
                printf("%d %10.5e\n",i,x[i]);
                }
        exit(EXIT_SUCCESS);
}
void inittab(double x[],int n)
        {
         unsigned int i;
        for (i=0;i<n;i++)
                x[i] = cos( (double) i *M_PI / (double) n);
}
```

Scanned with CamScanner

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
void inittab(double *,int );
int main(){
    double x[10];
    inittab(x,10);
    for(int i=0;i<10;i++)
        {
        printf("%d %10.5e\n",i,x[i]);
        }
    exit(EXIT_SUCCESS);
}
void inittab(double x[],int n)
    {
     unsigned int i;
    for (i=0;i<n;i++)
        x[i] = cos( (double) i *M_PI / (double) n);
}
```

Scanned with CamScanner

La spécification de type : const

- Si la fonction ne doit pas modifier les valeurs du tableau transmis dans les arguments de la fonction, une disposition provenant du C++ a été introduite en C pour sécuriser l'écriture des programmes.

- En plaçant la spécification de type const devant le type du tableau on oblige le compilateur à vérifier que le tableau ne sera pas modifié dans le sous-programme. Ainsi si on modifie le programme précédent, en ajoutant la fonction affichage on obtient :

Scanned with CamScanner

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
void inittab( double *,int );
void affiche (const double *,int);
int main(){
    const int n=10;
    double x[n];
    inittab(x,n);
    affiche(x,n);
    exit(EXIT_SUCCESS);
    }
void inittab( double x[], int n)
{
    for (int i=0;i<n;i++) x[i]=cos((double) i *M_PI/(double) n);
}
void affiche (const double x[],int n)
{
    for(int i=0;i<n;i++) printf("%d %10.5e\n",i ,x[i]);
}
```

Scanned with CamScanner

La spécification de type : const

- En plaçant la directive const à chaque fois que le tableau ne doit pas être modifié, on s'assure que la fonction n'introduira d'effets indésirables.

- La directive const peut être placée devant toute déclaration de variable dont on souhaite qu'elle ne soit pas modifiée dans la fonction (ou le programme principal) pendant l'exécution. Il est nécessaire d'initialiser la variable en même temps que celle-ci est déclarée.

```
const int n=10;
```

Scanned with CamScanner

Fonctions

- Il reste le cas des fonctions qui sont passées comme argument. Appelons la fonction appelante la fonction func_1 et la fonction appelée la func_2.
- Une manière simple consiste à définir un pointeur de la func_2 comme paramètre à passer à la fonction func_1.
- Ce pointeur est un peu particulier car il doit être déclaré avec le nombre et le type de variables de la func_2.
- Cela rend la syntaxe un peu délicate, mais le reste de l'écriture de la fonction func_1 reste simple. L'exemple suivant illustre le concept de pointeur d'une fonction

```
/* protopypes */
double func_2(double);
double func_1(double,double,long, double (*)(double));
/*
double func_1(double a, double b,long n, double (*f)(double));
*/
}
/* dans le programme, l'appel sera */
func_1(a,b,n,func_2);
```

Scanned with CamScanner

Université Sidi Mohammed Ben Abdellah
Ecole Nationale des Sciences Appliquées – Fès

STRUCTURES DE DONNÉES

Karima AISSAOUI
Karima.aissaoui1@usmba.ac.ma

2019/2020

Scanned with CamScanner

Pointeurs

- Définition, utilisation de base
- Utilisation avec les fonctions
- Utilisation avec les tableaux
- Utilisation avec les structures

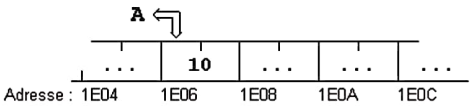
Scanned with CamScanner

Adressage

- Dans la programmation, nous utilisons des variables pour stocker des informations. La valeur d'une variable se trouve à un endroit spécifique dans la mémoire interne de l'ordinateur. Le nom de la variable nous permet alors d'accéder *directement* à cette valeur.
- ***Adressage direct:*** Accès au contenu d'une variable par le nom de la variable.

Exemple

```
short A;
A = 10;
```



Scanned with CamScanner

Adressage

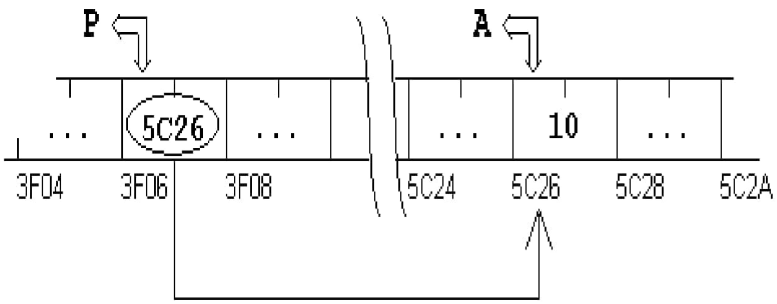
- Si nous ne voulons ou ne pouvons pas utiliser le nom d'une variable A, nous pouvons copier l'adresse de cette variable dans une variable spéciale P, appelée pointeur. Ensuite, nous pouvons retrouver l'information de la variable A en passant par le pointeur P.
- **Adressage indirect:** Accès au contenu d'une variable, en passant par un pointeur qui contient l'adresse de la variable.

Scanned with CamScanner

Adressage

Exemple

Soit A une variable contenant la valeur 10 et P un pointeur qui contient l'adresse de A. En mémoire, A et P peuvent se présenter comme suit:



Scanned with CamScanner

Définition: Pointeur

- Un pointeur est une variable spéciale qui peut contenir l'adresse d'une autre variable.
- En C, chaque pointeur est limité à un type de données. Il peut contenir l'adresse d'une variable simple de ce type ou l'adresse d'une composante d'un tableau de ce type.
- Si un pointeur P contient l'adresse d'une variable A, on dit que **'P pointe sur A'**.
- Remarque
- Les pointeurs et les noms de variables ont le même rôle: Ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur. Il faut quand même bien faire la différence:
  - * Un pointeur est une variable qui peut 'pointer' sur différentes adresses.
  - * Le nom d'une variable reste toujours lié à la même adresse.

Scanned with CamScanner

Les pointeurs

- Lors du travail avec des pointeurs, nous avons besoin
  - - d'un opérateur **'adresse de': &** pour obtenir l'adresse d'une variable.
  - - d'un opérateur **'contenu de':** * pour accéder au contenu d'une adresse.
  - - d'une syntaxe de déclaration pour pouvoir déclarer un pointeur.
- L'opérateur 'adresse de' : &
- **&<NomVariable>** fournit l'adresse de la variable <NomVariable>

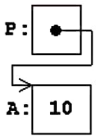
Scanned with CamScanner

Les pointeurs

- Soit P un pointeur non initialisé

- et A une variable (du même type) contenant la valeur 10 :

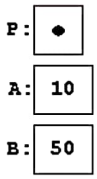
- Alors l'instruction
- ***P = &A;***
- affecte l'adresse de la variable A à la variable P. Dans notre représentation schématique, nous pouvons illustrer le fait que 'P pointe sur A' par une flèche:



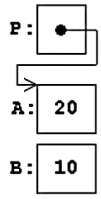
Scanned with CamScanner

Les pointeurs

- ***<NomPointeur>***
- désigne le contenu de l'adresse référencée par le pointeur <NomPointeur>
- Exemple
- Soit A une variable contenant la valeur 10, B une variable contenant la valeur 50 et P un pointeur non initialisé:



- Après les instructions,
- P = &A;
- B = *P;
- *P = 20;
- - P pointe sur A,
- - le contenu de A (référencé par *P) est affecté à B, et
- - le contenu de A (référencé par *P) est mis à 20.



Scanned with CamScanner

Déclaration et initialisation

- Déclaration d'un pointeur sur int

```
int * p;
```

- Convention d'initialisation avec NULL

```
p = NULL;
```

Scanned with CamScanner

Affectation

Affectation avec une adresse de variable

```
int a;
...
p = &a;
```

Scanned with CamScanner

Déréférencement

Déréférencement d'un pointeur

```
int b;
b = *p;
```

Scanned with CamScanner

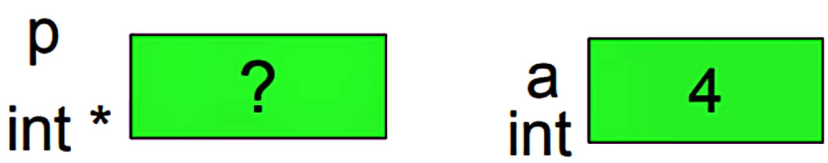
Exemple basique

```
int * p;
```

Scanned with CamScanner

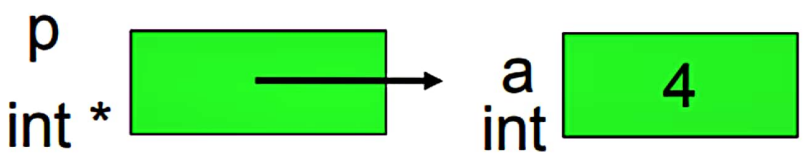
Exemple basique

```
int * p;
int a=4;
```



Scanned with CamScanner

```
int * p;
int a=4;
p = &a;
```



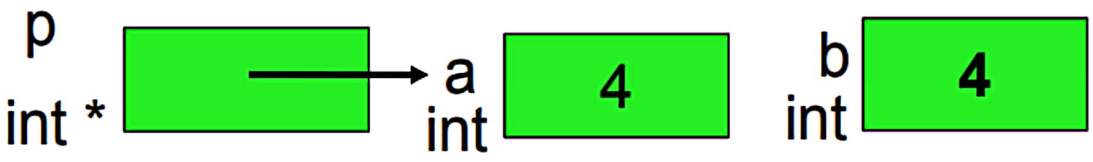
- `p` « pointe » sur `a`
- `&a` : adresse de `a`

Scanned with CamScanner

```
int * p;
int a=4;
p = &a;
int b;
```

Scanned with CamScanner

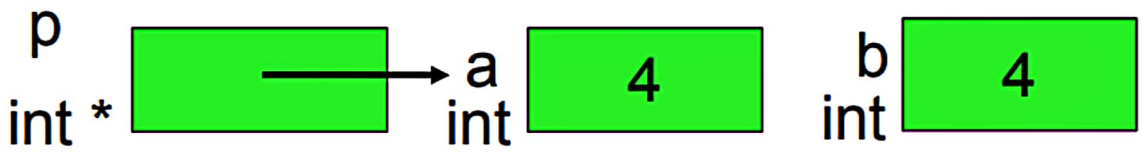
```
int * p;
int a=4;
p = &a;
int b;
b = *p;
```



- *p : contenu de la variable « pointée » par p
- *p : « déréférencement » de p

Scanned with CamScanner

```
int a=4; int * p = &a;
int b; b = *p;
```



- `p` « pointe » sur `a`
- `&a` adresse de `a`
- `*p` contenu de la variable pointée par `p` ou « déréférencement » de `p`

Scanned with CamScanner

```
float a=4.0; int * p = &a;
```

- Erreur de compilation: un pointeur est un pointeur sur des variables d'un type donné.

```
float a=4.0; float * p = &a;
```

- Une variable de type `truc *` pointe sur un `truc`

Scanned with CamScanner

- Pourquoi initialiser un pointeur avec `NULL` ?
- 3 cas: le pointeur pointe sur
  - la variable adéquate et on peut s'en servir
  - une variable inadéquate et le programme est incorrect
  - `NULL`
- Convention:
  - On affecte `NULL` quand un pointeur ne sert plus ou pas encore

Scanned with CamScanner

Avec cette convention, 2 cas:

- Pointeur sur variable adéquate
- Pointeur `NULL`

Avant d'utiliser un pointeur on teste sa valeur:

```
if (p!=NULL) { ... b = *p; ...}
else { ... }
```

Scanned with CamScanner

Exemple

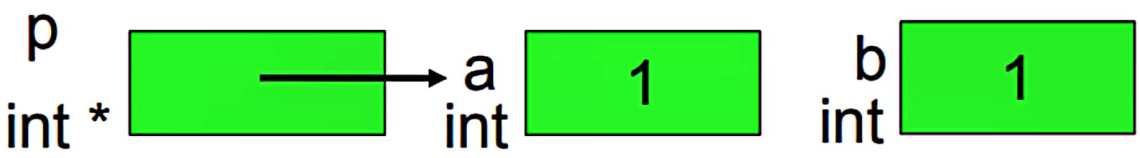
`printf("a = %d, b = %d, *p = %d.\n", a, b, *p);` après chaque ligne.

sortie du programme ?

```
int a = 1; int * p = &a; int b = *p;
a = 2;
b = 3;
p = &b;
a = 4;
b = 5;
```

Scanned with CamScanner

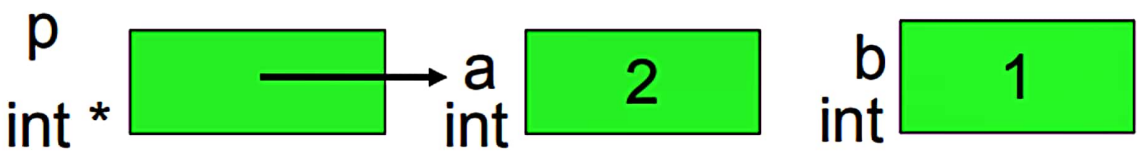
```
int a = 1; int * p = &a; int b = *p;
```




```
a = 1, b = 1, *p = 1.
```

Scanned with CamScanner

```
int a = 1; int * p = &a; int b = *p;
a = 2;
```




```
a = 2, b = 1, *p = 2.
```

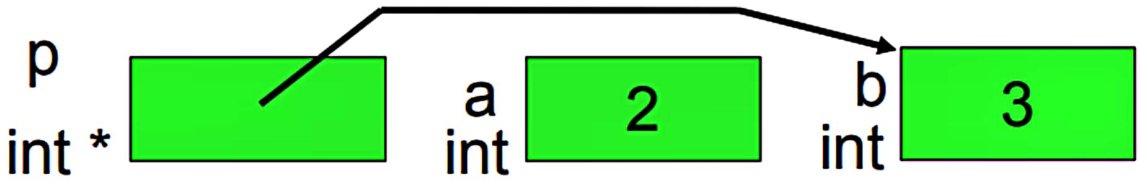
Scanned with CamScanner

```
int a = 1; int * p = &a; int b = *p;
a = 2;
b = 3;
```

a = 2, b = **3**, *p = 2.

Scanned with CamScanner

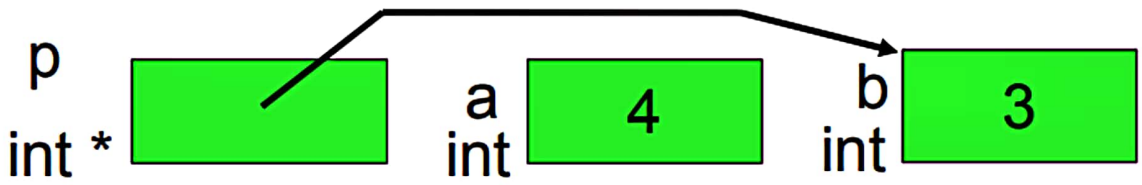
```
int a = 1; int * p = &a; int b = *p;
a = 2;
b = 3;
p = &b;
```



a = 2, b = 3, *p = **3**.

Scanned with CamScanner

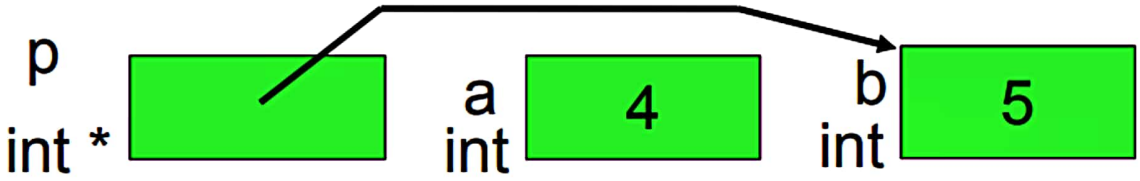
```
int a = 1; int * p = &a; int b = *p;
a = 2;
b = 3;
p = &b;
a = 4;
```



a = **4**, b = 3, *p = 3.

Scanned with CamScanner

```
int a = 1; int * p = &a; int b = *p;
a = 2;
b = 3;
p = &b;
a = 4;
b = 5;
```



a = 4, b = **5**, *p = **5**.

Scanned with CamScanner

LISTES CHAÎNÉES

Pr. Karima AISSAOUI

Karima.aissaoui1@usmba.ac.ma

Scanned with CamScanner

Introduction

- Le principe de base d'une structure de données, c'est de stocker des éléments auxquels le programmeur veut pouvoir accéder plus tard.
- On appelle les différentes utilisations possibles de la structure de données des *opérations*.
  - *lecture* : on veut récupérer un élément stocké dans la structure.
  - *insertion*: rajoute un nouvel élément dans la structure de données,
  - *suppression*, qui en enlève.
- Toutes les structures de données ne permettent pas les mêmes opérations, et surtout elles n'ont pas toujours le même *coût*.
  - Par exemple, sur certaines structures, il est très rapide d'ajouter un élément, dans d'autres c'est difficile et cela peut demander une réorganisation complète.

Scanned with CamScanner

Tableaux

- Le tableau est sans doute la structure de données la plus courante, du moins dans les langages dérivés ou inspirés par le langage C.
- Le principe d'un tableau est très simple : on stocke les éléments dans des cases, chaque case étant étiquetée d'un numéro (appelé *indice*). Pour accéder à un élément particulier d'un tableau, on donne son indice.
- Les indices sont des entiers consécutifs qui commencent à 0. En particulier, si n est la taille du tableau, le dernier élément se trouve à l'indice n-1.
- Demander l'accès à un indice qui n'existe pas provoque une erreur.

Scanned with CamScanner

Listes chaînées

- Une liste est un ensemble d'objets de même type constituant les éléments de la liste.
- Les éléments sont chaînés entre eux et on peut facilement ajouter ou extraire un ou plusieurs éléments.
- Une liste est une structure de données telle que chaque élément contient :
  - des informations sur l'objet
  - un pointeur sur un autre élément de la liste, ou un pointeur NULL s'il n'y a pas d'élément suivant.

Scanned with CamScanner

Listes chaînées

- Un élément ou une cellule est un ensemble de cases qui permettent de stocker des données, et auxquelles on accède par leur nom.
- Par exemple, on pourra décrire une structure servant à stocker l'adresse et le numéro d'une personne comme une cellule à trois
champs, nom, adresse et téléphone.
- En langage C, une cellule est de la forme struct.

Scanned with CamScanner

Listes chaînées

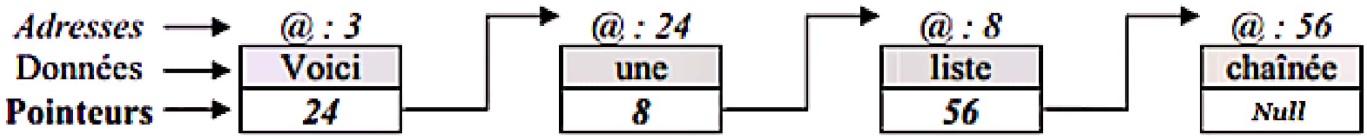
- La création (et la destruction) d'une cellule, ainsi que la lecture ou la modification d'un champ, se fait en temps constant.

➔ Une liste est :

- soit la liste vide ;
- constituée d'éléments reliés entre eux par des pointeurs.

Scanned with CamScanner

Listes chaînées



- Le 1er élément de la liste vaut « Voici » à l'adresse 3 (début de la liste chaînée)
- Le 2ème élément de la liste vaut « une » à l'adresse 24 (car le pointeur de la cellule d'adresse 3 est égal à 24)
- Le 3ème élément de la liste vaut « liste » à l'adresse 8 (car le pointeur de la cellule d'adresse 24 est égal à 8)
- Le 4ème élément de la liste vaut « chainée » à l'adresse 56 (car le pointeur de la cellule d'adresse 8 est égal à 56).

Scanned with CamScanner

Listes chaînées

- Autrement dit, une *liste* est
  - soit vide,
  - soit un élément suivi d'une *liste*
- Une liste peut donc être :
  - la liste vide (0 élément),
  - ou un élément suivi de la liste vide (1 élément),
  - ou un élément suivi d'un élément suivi de la liste vide (2 éléments), etc.

Scanned with CamScanner

Listes chaînées

- On dit que l'élément dans le champ tête est la tête de la liste, et le reste est sa queue.
- La queue d'une liste contient tous les éléments de la liste, sauf le premier.
- Par exemple, la queue de la liste [1; 2; 3] est [2; 3].
- Il existe différents types de listes chaînées :
  - **Liste chaînée simple** constituée d'éléments reliés entre eux par des pointeurs.
  - **Liste chaînée ordonnée** où l'élément suivant est plus grand que le précédent. L'insertion et la suppression d'élément se font de façon à ce que la liste reste triée.
  - **Liste doublement chaînée** où chaque élément dispose non plus d'un mais de deux pointeurs pointant respectivement sur l'élément précédent et l'élément suivant. Ceci permet de lire la liste dans les deux sens, du premier vers le dernier élément ou inversement.
  - **Liste circulaire** où le dernier élément pointe sur le premier élément de la liste. S'il s'agit d'une liste doublement chaînée alors de premier élément pointe également sur le dernier.

Scanned with CamScanner

Implémentation

- En C, il faut construire la liste chainée soi-même.
- On utilise une représentation très classique :
  - une structure pour les cellules,
  - et le pointeur NULL pour la liste vide.

```c
#include <stdlib.h>

typedef struct Element Element;
struct Element
{
    int val;
    Element *p_suivant;
};
```

  - val est le contenu de la cellule, (ici, un entier)
  - p_suivant pointe vers la cellule suivante, ou vaut NULL (fin de liste).

Scanned with CamScanner

Initialisation

- Déclarations de 3 listes chainées de façons différentes mais équivalentes.

```
typedef Element* llist;

int main(int argc, char **argv)
{
    llist ma_liste1=NULL;
    Element *ma_liste2=NULL;
    struct Element *ma_liste3=NULL;

    return 0;

}
```

Scanned with CamScanner

Implémentation

- Pour pouvoir sauvegarder certains éléments spécifiques de la liste tels que la taille, le premier élément, il faut créer un nouveau type de structure :

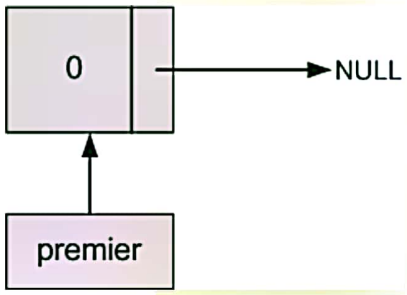
```
typedef struct Liste Liste;
struct Liste
{
    short int  longueur;
    Element *p_tete;
} ;
```

- Il est bien sûr possible d'en ajouter d'autre, tel qu'un pointeur sur le dernier élément ou l'élément courant de la liste.

Scanned with CamScanner

Initialisation

```
Liste *initialisation(int valeur)
{
    Liste *liste = malloc(sizeof(*liste));
    Element *element = malloc(sizeof(*element));

    if (liste == NULL || element == NULL)
    {
        exit(EXIT_FAILURE);
    }

    element->val = valeur;
    element->p_suivant = NULL;

    liste->longueur=1;
    liste->p_tete = element;

    return liste;
}
```



Scanned with CamScanner

Ajout/ Retrait/ Affichage

Scanned with CamScanner

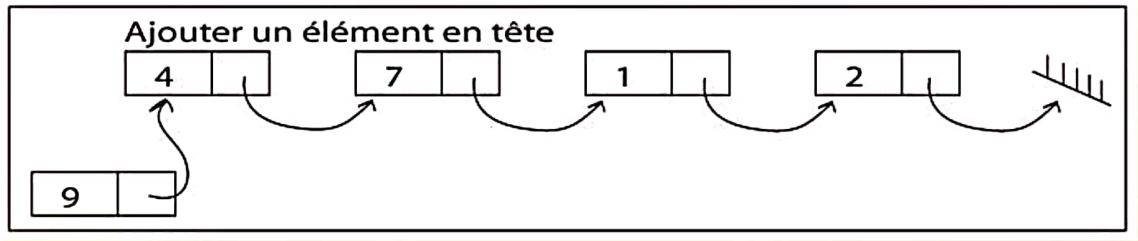
Ajouter un élément

- Lorsque nous voulons ajouter un élément dans une liste chaînée, il faut savoir où l'insérer.
- Les deux ajouts génériques des listes chaînées sont:
  - les ajouts en tête,
  - et les ajouts en fin de liste.
- Nous allons étudier ces deux moyens d'ajouter un élément à une liste.

Scanned with CamScanner

Ajouter en tête

- Lors d'un ajout en tête,
  1. nous allons créer un élément,
  2. lui assigner la valeur que l'on veut ajouter,
  3. raccorder cet élément à la liste passée en paramètre.
  4. Lors d'un ajout en tête, on devra donc assigner à p_suivant l'adresse du premier élément de la liste passé en paramètre.
- Visualisons tout ceci sur un schéma :



Scanned with CamScanner

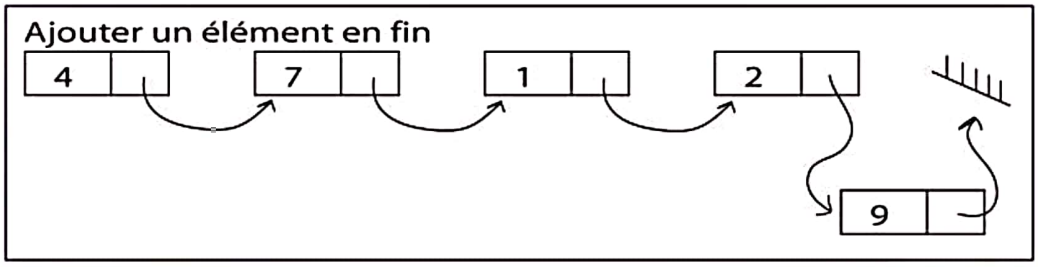
Ajouter en tête

```
void insertion(Liste *liste, int nvNombre)
{
    /* Création du nouvel élément */
    Element *nouveau = malloc(sizeof(*nouveau));
    if (liste == NULL || nouveau == NULL)
    {
        exit(EXIT_FAILURE);
    }
    nouveau->val = nvNombre;

    /* Insertion de l'élément au début de la liste */
    nouveau->p_suivant = liste->p_tete;
    liste->p_tete = nouveau;

    liste->longueur=liste->longueur+1;
}
```

Scanned with CamScanner

Ajouter en fin de liste(Exercice)

1. Il faut tout d'abord créer un nouvel élément,
2. lui assigner sa valeur,
3. et mettre l'adresse de l'élément suivant à NULL.
4. il faut faire pointer le dernier élément de la liste originale sur le nouvel élément que nous venons de créer. Pour ce faire,
    1. il faut créer un pointeur temporaire sur Element qui va se déplacer d'élément en élément,
    2. et vérifier si cet élément est le dernier de la liste. Un élément sera forcément le dernier de la liste si NULL est assigné à son champ p_suivant.



Scanned with CamScanner

Afficher les éléments de la liste

```
void afficherListe(Liste *liste)
{
    if (liste == NULL)
    {
        exit(EXIT_FAILURE);
    }

    Element *actuel = liste->p_tete;

    while (actuel != NULL)
    {
        printf("%d -> ", actuel->val);
        actuel = actuel->p_suivant;
    }
    printf("NULL\n");
}
```

Scanned with CamScanner

Supprimer le 1er élément de la liste

```
void suppression(Liste *liste)
{
    if (liste == NULL)
    {
        exit(EXIT_FAILURE);
    }

    if (liste->p_tete != NULL)
    {
        Element *aSupprimer = liste->p_tete;
        liste->p_tete = liste->p_tete->p_suivant;
        liste->longueur=liste->longueur-1;
        free(aSupprimer);
    }
}
```

Scanned with CamScanner

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "liste_chainee.h"


int main()
{
    Liste *maListe = initialisation(0);

    insertion(maListe, 4);
    insertion(maListe, 8);
    insertion(maListe, 15);
    suppression(maListe);

    afficherListe(maListe);

    return 0;
}
```

```
8 -> 4 -> 0 -> NULL

Process returned 0 (0x0)   execution time : 0.570 s
Press any key to continue.
```

Scanned with CamScanner

LISTES DOUBLEMENT CHAÎNÉES

Pr. Karima AISSAOUI

Karima.aissaoui1@usmba.ac.ma

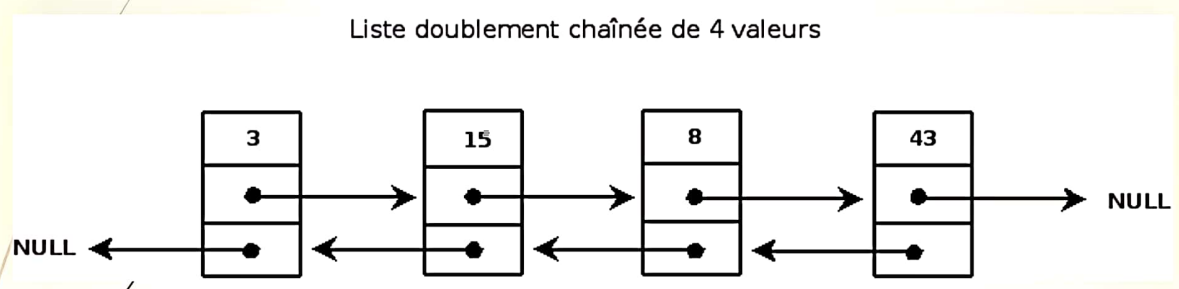
Scanned with CamScanner

Définition

- Lorsque chaque élément d'une liste chainée pointe vers l'élément suivant, nous parlons de liste **simplement** chainée.
- Lorsque chaque élément d'une liste pointe à la fois vers l'élément suivant et précédent, nous parlons alors de liste **doublement** chainée.

Scanned with CamScanner

Définition



- chaque élément d'une liste doublement chainée contient :
  - Une donnée (ici un simple entier)
  - Un pointeur vers l'élément suivant (NULL si l'élément suivant n'existe pas)(ie le dernier élément)
  - Un pointeur vers l'élément précédent (NULL si l'élément précédent n'existe pas)(ie le premier élément)

Scanned with CamScanner

Implémentation

- La première structure permet de représenter un 'node' (élément) de la liste chaînée.

```
struct node
{
    int data;
    struct node *p_next;
    struct node *p_prev;
};
```

- chaque élément de la liste contient:
  - un élément de type **int**.
  - **p_next** pointe vers l'élément suivant (ou NULL s'il s'agit du dernier élément de la liste)
  - **p_prev** pointe vers l'élément précédent (ou NULL s'il s'agit du premier élément)

Scanned with CamScanner

Implémentation

- Pour représenter la liste chaînée, nous utiliserons une deuxième liste :

```
typedef struct dlist
{
    short int length;
    struct node *p_tail;
    struct node *p_head;
} Dlist;
```

Scanned with CamScanner

Manipulation d'une liste doublement chainée

Scanned with CamScanner

Allouer une nouvelle liste

- p_new est la nouvelle liste.
- malloc sert à réserver de l'espace mémoire pour cette liste.

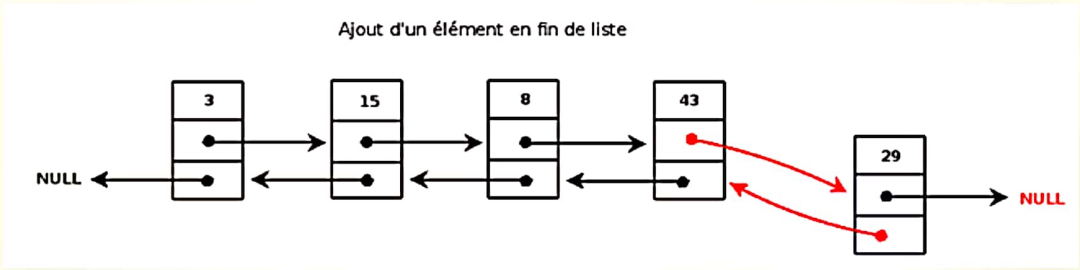
```
Dlist *dlist_new(void)
{
    Dlist *p_new = malloc(sizeof *p_new);
    if (p_new != NULL)
    {
        p_new->length = 0;
        p_new->p_head = NULL;
        p_new->p_tail = NULL;
    }
    return p_new;
}
```

Scanned with CamScanner

Ajout en fin de liste

- Vu que nous avons un pointeur vers la fin de la liste,

➔ nous n'avons donc nul besoin de parcourir la liste en entier afin d'arriver au dernier élément, nous l'avons déjà.



Scanned with CamScanner

Ajout en fin de liste

1. Tout d'abord, il faut vérifier que la liste n'est pas NULL.
2. Si elle ne l'est pas, nous allons créer un nouvel élément (nouveau node). Une fois celui-ci créé,
   - nous devons stocker la donnée dans le champ donnée (data)
   - puis faire pointer **p_next** vers NULL car ce sera le dernier élément de la liste.
3. A partir de là, deux possibilités s'offrent :
   - S'il n'existe pas de dernier élément (donc la liste est vide) Alors
     - Nous faisons pointer **p_prev** vers NULL
     - Nous faisons pointer la tête et la fin de liste vers notre nouvel élément
   - Sinon
     - Nous rattachons le dernier élément de notre liste à notre nouvel élément
     - Nous faisons pointer **p_prev** vers le dernier élément de notre liste
     - Nous faisons pointer notre fin de liste vers notre nouvel élément
4. Enfin, nous incrémentons notre champ length de notre liste puis nous retournons la liste. Tout ceci constitue alors l'algorithme d'ajout en fin de liste.

Scanned with CamScanner

Ajout en fin de liste

```
Dlist *dlist_append(Dlist *p_list, int data)
{
    if (p_list != NULL) /* On vérifie si la liste a été allouée */
    {
        struct node *p_new = malloc(sizeof *p_new); /* Création d'un nouveau node */
        if (p_new != NULL) /* On vérifie si le malloc n'a pas échoué */
        {
            p_new->data = data; /* On 'enregistre' la donnée */
            p_new->p_next = NULL; /* On fait pointer p_next vers NULL */
            if (p_list->p_tail == NULL) /* Cas où la liste est vide
                                           (pointeur vers fin de liste à  NULL) */
            {
                p_new->p_prev = NULL; /* On fait pointer p_prev vers NULL */
                p_list->p_head = p_new; /* On fait pointer la tête de liste vers le nouvel élément */
                p_list->p_tail = p_new; /* On fait pointer la fin de liste vers le nouvel élément */
            }
            else /* Cas où des éléments sont déjà présents dans la liste */
            {
                p_list->p_tail->p_next = p_new; /* On relie le dernier élément de la liste vers le nouvel élément */
                p_new->p_prev = p_list->p_tail; /* On fait pointer p_prev vers le dernier élément de la liste */
                p_list->p_tail = p_new; /* On fait pointer la fin de liste vers
                                           le nouvel élément (fin du chaînage: 3 étapes) */
            }
            p_list->length++; /* Incrémentation de la taille de la liste */
        }
    }
    return p_list; /* on retourne la nouvelle liste */
}
```

Scanned with CamScanner

Ajout en début de liste

- Pour ajouter un élément en début de liste, il faut utiliser exactement le même procédé que pour l'ajout en fin de liste.
- Grâce aux pointeurs en début et en fin de liste, il est possible de reprendre les mêmes implémentations.

Scanned with CamScanner

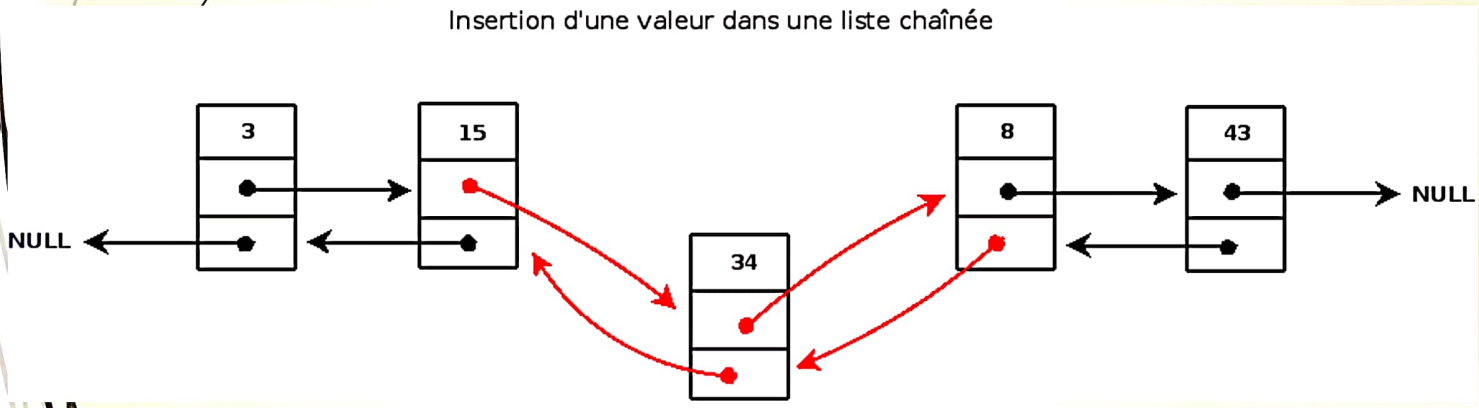
Insérer un élément

- Maintenant, si l'on désire ajouter un élément n'importe où dans notre liste ?
- Tout d'abord, nous aurons besoin de parcourir notre liste.
- Nous aurons aussi besoin d'un compteur (que l'on nommera) « i » afin de nous arrêter à la position où nous souhaitons insérer notre nouvel élément.
- Il faut alors réfléchir des différents cas de figure qui peuvent intervenir lorsque nous aurons trouvé notre position:
  - Soit nous sommes en fin de liste
  - Soit nous sommes en début de liste
  - Soit nous sommes en milieu de liste
- Cependant, les deux premiers cas sont très faciles à traiter. Enfin, nous disposons de fonctions permettant d'ajouter un élément en début et en fin de liste, il nous suffit donc de les réaliser(appeler).
- Il faut alors gérer le cas où nous nous trouvons en milieu de liste.

Scanned with CamScanner

Insérer un élément

- Voici un petit schéma permettant de mieux cerner la situation:



Scanned with CamScanner

Insérer un élément

- Il faut tout d'abord relier les éléments suivants et précédents au nouvel élément
- puis, inversement, il faut relier le nouvel élément aux éléments suivant et précédent.
- A noter qu'il sera nécessaire d'avoir préalablement créer un nouvel élément sans quoi le chaînage ne pourra pas avoir lieu.
- Le chaînage va alors se dérouler en 4 étapes.
  - Pour parcourir la liste, il faut récupérer le pointeur vers le début de liste dans un pointeur temporaire.
  - C'est ce pointeur temporaire qui servira à parcourir la liste.
  - Schématiquement, la liste sera parcourue de gauche à droite.
  - Le compteur sera bien évidemment incrémenté lors du parcours de chaque maillon de la liste.

Scanned with CamScanner

```
Dlist *dlist_insert(Dlist *p_list, int data, int position) {
    if (p_list != NULL) {
        struct node *p_temp = p_list->p_head;
        int i = 1;
        while (p_temp != NULL && i <= position) {
            if (position == i) {
                if (p_temp->p_next == NULL) {
                    p_list = dlist_append(p_list, data);
                }
                else if (p_temp->p_prev == NULL) {
                    p_list = dlist_prepend(p_list, data);
                }
                else {
                    struct node *p_new = malloc(sizeof *p_new);
                    if (p_new != NULL) {
                        p_new->data = data;
                        p_temp->p_next->p_prev = p_new;
                        p_new=>p_next=p_temp=>p_next;
                        p_temps->p_next=p_new;
                        p_new->p_prev=p_temp;
                        p_list->length++;
                    }
                }
            }
            else
            {
                p_temp = p_temp->p_next;
            }
            i++;
        }
    }
    return p_list;
}
```

Scanned with CamScanner

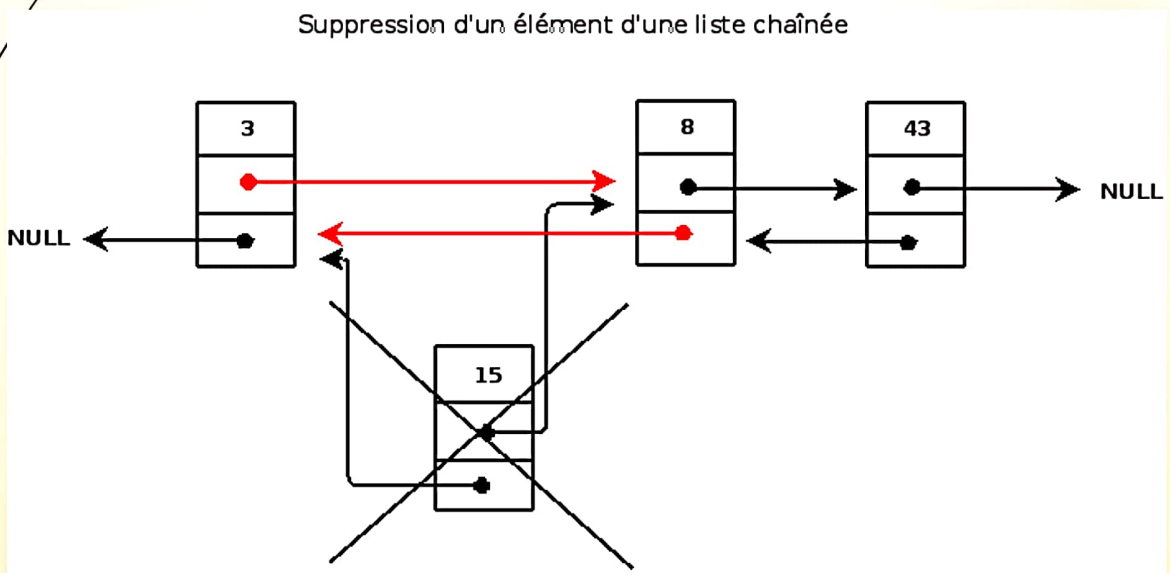
Insérer un élément

- Pour parcourir la liste,
  - nous utilisons un pointeur nommé **p_temp**. Au tout début, celui-ci pointe vers le premier élément de notre liste (**p_list->p_head**).
  - nous utilisons une structure de type while.
  - Tant que nous n'avons pas atteint la fin de liste **(p_temp != NULL)** ***et*** tant que nous ne sommes pas à la position où nous voulons insérer notre élément **(position <= i),** nous bouclons.
- Dès lors que nous avons atteint notre position (position == i), nous devons alors effectuer nos trois tests :
  - Si nous sommes en fin de liste (p_temp->p_next == NULL), nous utilisons la fonction d'ajout en fin de liste
  - Sinon, si nous sommes en début de liste (p_temp->p_prev == NULL), nous utilisons la fonction en début de liste
  - Sinon, il faut créer un nouvel élément et réaliser le chaînage sans oublier de stocker la donnée dans le champ data
- Enfin, si nous n'avons pas encore atteint notre position, nous passons à l'élément suivant (p_temp = p_temp->p_next).

Scanned with CamScanner

Supprimer un élément selon sa valeur

- Supprimer un élément en fonction de sa valeur.
- Voici un schéma pour mieux visualiser le procédé:



Scanned with CamScanner

Supprimer un élément selon sa valeur

- Ici, nous décidons de supprimer l'élément portant la valeur 15.
- Tout d'abord, il faut parcourir la liste à la rechercher de l'élément à supprimer. Dès que l'on aura trouver la valeur correspondante, trois possibilités s'offriront à nous:
  - l'élément se trouve en fin de liste
  - l'élément se trouve en début de liste
  - l'élément se trouve en milieu de liste

Scanned with CamScanner

Supprimer un élément selon sa valeur

- Si l'élément se trouve en fin de liste, Alors il faut faire pointer notre p_tail vers l'avant dernier élément et faire pointer le pointeur vers l'élément suivant de l'avant dernier élément vers NULL.
- Sinon, si l'élément se trouve en début de liste, Alors il faudra faire pointer notre p_head vers le second élément et faire pointer le pointeur vers l'élément précédent du second élément vers NULL.
- Sinon, il faudra relier l'élément précédent à l'élément que l'on veut supprimer vers l'élément suivant à l'élément que l'on veut supprimer et il faudra aussi relier l'élément suivant à l'élément su l'on veut supprimer vers l'élément précédent à l'élément que l'on veut supprimer.

- Une fois ceci fait, il ne nous restera plus qu'à supprimer notre élément trouver et à décrémenter la taille de notre liste.
- A noter cependant que nous utilisons une variable supplémentaire nommée « found » pour nous arrêter au premier élément trouvé. Lorsque l'élément est trouvé, cette variable change d'état et prend la valeur « 1 », marquant ainsi l'arrêt de la boucle de parcours.

Scanned with CamScanner

```
Dlist *dlist_remove(Dlist *p_list, int data)
{
    if (p_list != NULL)
    {
        struct node *p_temp = p_list->p_head;
        int found = 0;
        while (p_temp != NULL && !found)
        {
            if (p_temp->data == data)
            {
                if (p_temp->p_next == NULL)
                {
                    p_list->p_tail = p_temp->p_prev;
                    p_list->p_tail->p_next = NULL;
                }
                else if (p_temp->p_prev == NULL)
                {
                    p_list->p_head = p_temp->p_next;
                    p_list->p_head->p_prev = NULL;
                }
                else
                {
                    p_temp->p_next->p_prev = p_temp->p_prev;
                    p_temp->p_prev->p_next = p_temp->p_next;
                }
                free(p_temp);
                p_list->length--;
                found = 1;
            }
            else
            {
                p_temp = p_temp->p_next;
            }
        }
```

Scanned with CamScanner

Supprimer un élément selon sa position(Exercice)

- Nous allons créer une fonction permettant de supprimer le $n^{ème}$ élément d'une liste doublement chaînée.
- Il s'agit exactement du même procédé utilisé précédemment, mis à part le fait que nous aurons besoin d'une variable supplémentaire nous permettant de stocker la position à laquelle nous nous trouvons.

Scanned with CamScanner

Recherche un élément selon sa valeur

- Nous allons parcourir notre liste tant que nous n'aurons pas trouvé notre élément (variable found).
- Si nous trouvons notre élément, nous utilisons alors les fonctions déjà à notre disposition (dlist_new et dlist_append) pour créer notre liste qui sera retourné.

Scanned with CamScanner

```
Dlist *dlist_find(Dlist *p_list, int data)
{
    Dlist *ret = NULL;
    if (p_list != NULL)
    {
        struct node *p_temp = p_list->p_head;
        int found = 0;
        while (p_temp != NULL && !found)
        {
            if (p_temp->data == data)
            {
                ret = dlist_new();
                ret = dlist_append(ret, data);
                found = 1;
            }
            else
            {
                p_temp = p_temp->p_next;
            }
        }
    }
    return ret;
}
```

Scanned with CamScanner

Recherche un ensemble d'éléments selon une même valeur

- Il faut créer une liste.
- Si on trouve un élément, on l'initialise.
- Puis ajouter les éléments trouvés dedans

Scanned with CamScanner

```c
Dlist *dlist_find_all(Dlist *p_list, int data)
{
    Dlist *ret = NULL;
    if (p_list != NULL)
    {
        struct node *p_temp = p_list->p_head;
        while (p_temp != NULL)
        {
            if (p_temp->data == data)
            {
                if (ret == NULL)
                {
                    ret = dlist_new();
                }
                ret = dlist_append(ret, data);
            }
            p_temp = p_temp->p_next;
        }
    }
    return ret;
}
```

Scanned with CamScanner

Libérer une liste

- Après avoir utilisé la liste, il faut **libérer** tous les éléments alloués sous peine d'obtenir ce que l'on nomme des **fuites de mémoire** (*leak memory*).

```
void dlist_delete(Dlist **p_list)
{
    if (*p_list != NULL)
    {
        struct node *p_temp = (*p_list)->p_head;
        while (p_temp != NULL)
        {
            struct node *p_del = p_temp;
            p_temp = p_temp->p_next;
            free(p_del);
        }
        free(*p_list);
        *p_list = NULL;
    }
}
```

Scanned with CamScanner

Libérer une liste

- Dans cette fonction, nous utilisons un **double pointeur**. En effet, notre fonction delete doit directement effectuer les modifications sur la liste.
- Autrement dit, celle-ci doit faire des modifications sur un objet de type Dlist *.
- En premier lieu, nous vérifions si la liste que nous avons récupérée n'est pas NULL.
- Il faut parcourir ensuite chaque élément de la liste (à noter la présence de parenthèses afin de résoudre la priorité de l'opérateur -> sur l'opérateur *).
- Seulement, nous prenons garde de sauvegarder l'élément courant dans un pointeur p_del (qui sera l'élément que nous supprimerons).
- En effet, si nous n'utilisons pas de pointeur intermédiaire, lorsque nous supprimerons notre premier élément, p_temp->p_next n'existera plus puisque notre élément aura été supprimé, nous aurons donc un plantage.
- ➔ C'est pour cela que nous sauvegardons d'abord notre élément courant, puis nous passons à l'élément suivant et enfin nous supprimons notre élément courant (p_del).
- Quand tous les éléments sont supprimés, nous terminons par supprimer notre liste puis nous la remettons à NULL.

Scanned with CamScanner

Pile

Pr. Karima AISSAOUI

Karima.aissaoui1@usmba.ac.ma

Scanned with CamScanner

Introduction

- Les listes chainées sont particulièrement flexibles car on peut insérer et supprimer des données à n'importe quel endroit, à n'importe quel moment.
- Les piles et les files que nous allons découvrir ici sont deux variantes un peu particulières des listes chaînées.
- Elles permettent de contrôler la manière dont sont ajoutés les nouveaux éléments. Cette fois, on ne va plus insérer de nouveaux éléments au milieu de la liste mais **seulement au début ou à la fin.**

Scanned with CamScanner

Piles et Files

- Les piles et les files sont très utiles pour des programmes qui doivent traiter des données qui arrivent au fur et à mesure.
- Commençons par les piles:

Scanned with CamScanner

Les piles

- Imaginez une pile de pièces. Vous pouvez ajouter des pièces une à une en haut de la pile, mais aussi en enlevant depuis le haut de la pile. Il est en revanche impossible d'enlever une pièce depuis le bas de la pile.

Scanned with CamScanner

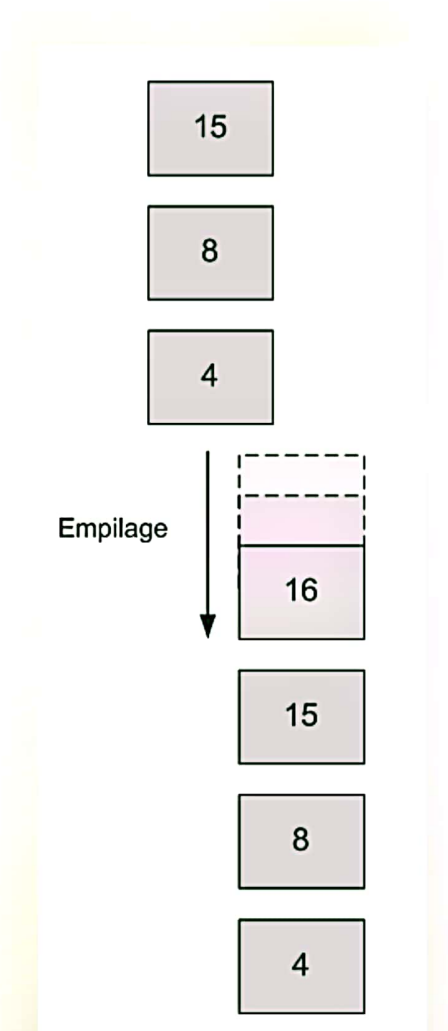
Fonctionnement des piles

- Le principe des piles en programmation est :
  - de stocker des données
    - au fur et à mesure
    - les unes au-dessus des autres
  - pour pouvoir les récupérer plus tard.

Scanned with CamScanner

Empilage

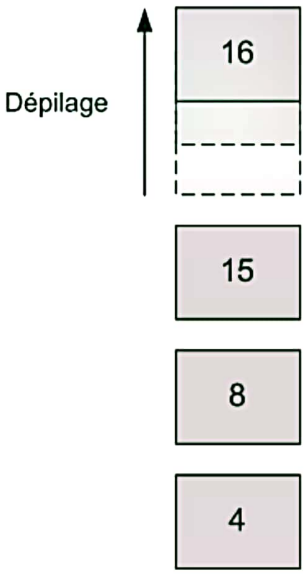
- Si j'ajoute un élément
- (on parle d'**empilage**),
- il sera placé au-dessus,
- comme dans Tetris



Scanned with CamScanner

Dépilage

- On récupère les données une à une, en commençant par la dernière qui vient d'être posée tout en haut de la pile.
- On enlève les données au fur et à mesure, jusqu'à la dernière tout en bas de la pile.



Scanned with CamScanner

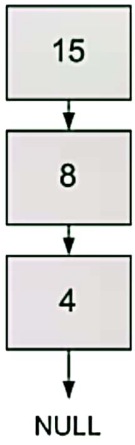
Fonctionnement des piles

- On dit que c'est un algorithme **LIFO**, ce qui signifie « Last In First Out ». « Le dernier élément qui a été ajouté est le premier à sortir ».
- Les éléments de la pile sont reliés entre eux à la manière d'une liste chaînée.
- Ils possèdent un pointeur vers l'élément suivant
- Ils ne sont donc pas forcément placés côte à côte en mémoire.

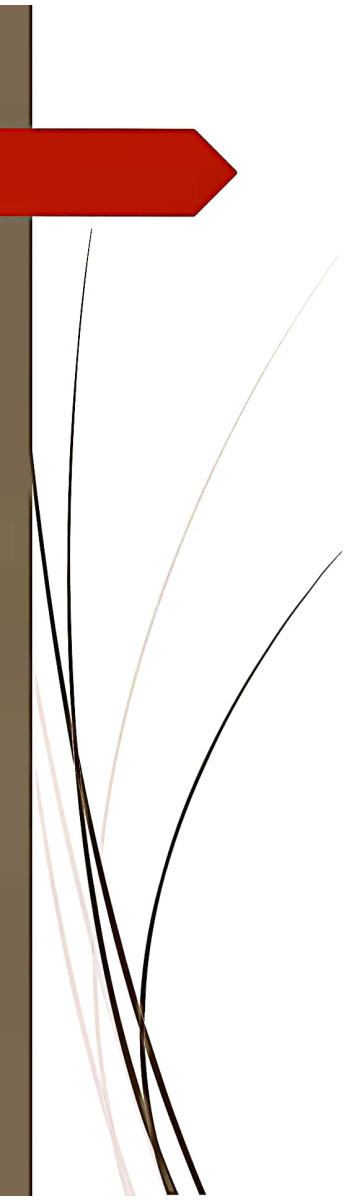
Scanned with CamScanner

Fonctionnement des piles

- Le dernier élément (tout en bas de la pile) doit pointer vers NULL pour indiquer qu'on a… touché le fond.



Scanned with CamScanner

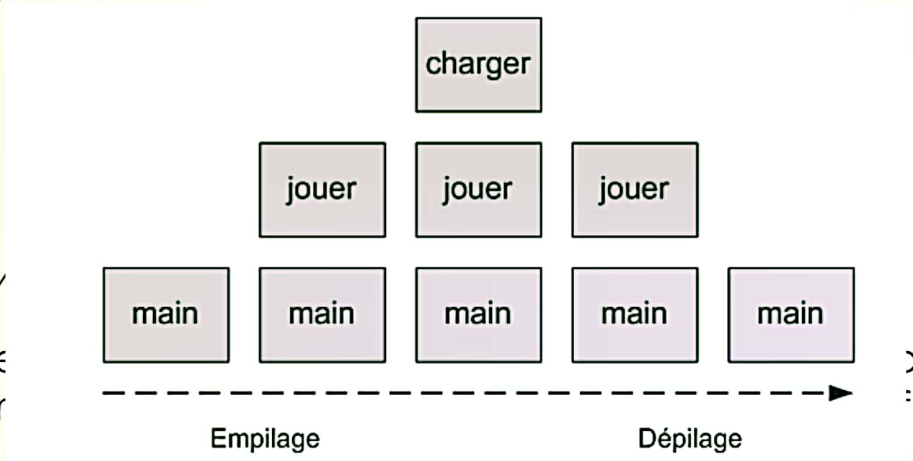


À quoi est-ce que tout cela peut bien servir, concrètement ?

- Un programme commence par la fonction main ;
  1. vous y appelez la fonction jouer ;
  2. cette fonction jouer fait appel à son tour à la fonction charger ;
  3. une fois que la fonction charger est terminée, on retourne à la fonction jouer ;
  4. une fois que la fonction jouer est terminée, on retourne au main ;
  5. enfin, une fois le main terminé, il n'y a plus de fonction à appeler, le programme s'achève.

Scanned with CamScanner

Pour « retenir » l'ordre dans lequel les fonctions ont été appelées, l'ordinateur crée une pile de ces fonctions au fur et à mesure.



- Grâce … onction il doit retour… faut, il retrouvera toujou…

Scanned with CamScanner

Création d'un système de pile

- Chaque élément de la pile aura une structure identique à celle d'une liste chaînée :

```
typedef struct Element Element;
struct Element
{
    int nombre;
    Element *suivant;
};
```

Scanned with CamScanner

Création d'un système de pile

- La structure de contrôle contiendra l'adresse du premier élément de la pile, celui qui se trouve tout en haut :

```
typedef struct Pile Pile;
struct Pile
{
    Element *premier;
};
```

Scanned with CamScanner

Création d'un système de pile

- Contrairement aux listes chaînées, on ne parle pas d'ajout ni de suppression.
- On parle d'**empilage** et de **dépilage**.
- Ainsi, on ne peut ajouter et retirer un élément qu'en haut de la pile.
- Nous aurons besoin en tout et pour tout des fonctions suivantes :
  - empilage d'un élément ;
  - dépilage d'un élément.
  - affichage de la pile.

Scanned with CamScanner

Empilage

- La fonction « empiler » doit prendre en paramètre la structure de contrôle de la pile (de type Pile) ainsi que le nouveau nombre à stocker.
- Nous stockons ici des int, mais rien n'empêche d'adapter ces exemples avec un autre type de données.
- On peut stocker n'importe quoi :
  - des double,
  - des char,
  - des chaînes,
  - des tableaux
  - ou même d'autres structures !

Scanned with CamScanner

```c
void empiler(Pile *pile, int nvNombre)
{
    Element *nouveau = malloc(sizeof(*nouveau));
    if (pile == NULL || nouveau == NULL)
    {
        exit(EXIT_FAILURE);
    }

    nouveau->nombre = nvNombre;
    nouveau->suivant = pile->premier;
    pile->premier = nouveau;
}
```

Scanned with CamScanner

Empilage

- L'ajout se fait en début de pile car il est impossible de le faire au milieu d'une pile.
- C'est le principe même de son fonctionnement, on ajoute toujours par le haut.
- De ce fait, contrairement aux listes chaînées, on ne doit pas créer de fonction pour insérer un élément au milieu de la pile. Seule la fonction empiler permet d'ajouter un élément.

Scanned with CamScanner

Dépilage

- Le rôle de la fonction de dépilage est:
  - de supprimer l'élément tout en haut de la pile.
  - de retourner l'élément qu'elle dépile.
- On demande toujours à récupérer le premier.

➔ On ne parcourt pas la pile pour aller y chercher le second ou le troisième élément.

Scanned with CamScanner

Dépilage

- Notre fonction « depiler » va donc retourner un int correspondant au nombre qui se trouvait en tête de

```
int depiler(Pile *pile)
{
    if (pile == NULL)
    {
        exit(EXIT_FAILURE);
    }

    int nombreDepile = 0;
    Element *elementDepile = pile->premier;

    if (pile != NULL && pile->premier != NULL)
    {
        nombreDepile = elementDepile->nombre;
        pile->premier = elementDepile->suivant;
        free(elementDepile);
    }

    return nombreDepile;
}
```

Scanned with CamScanner

Dépilage

1. On récupère le nombre en tête de pile pour le renvoyer à la fin de la fonction.
2. On modifie l'adresse du premier élément de la pile, puisque celui-ci change.
3. Enfin, on supprime l'ancienne tête de pile grâce à free.

Scanned with CamScanner

Affichage de la pile

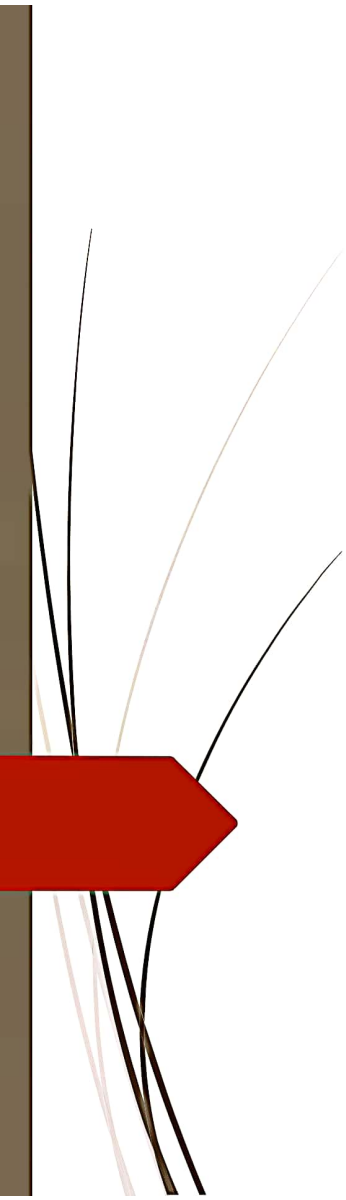
```
void afficherPile(Pile *pile)
{
    if (pile == NULL)
    {
        exit(EXIT_FAILURE);
    }
    Element *actuel = pile->premier;

    while (actuel != NULL)
    {
        printf("%d\n", actuel->nombre);
        actuel = actuel->suivant;
    }

    printf("\n");
}
```

Scanned with CamScanner

Main.C

```c
int main()
{
    Pile *maPile = initialiser();

    empiler(maPile, 4);
    empiler(maPile, 8);
    empiler(maPile, 15);
    empiler(maPile, 16);
    empiler(maPile, 23);
    empiler(maPile, 42);

    printf("\nEtat de la pile :\n");
    afficherPile(maPile);

    printf("Je depile %d\n", depiler(maPile));
    printf("Je depile %d\n", depiler(maPile));

    printf("\nEtat de la pile :\n");
    afficherPile(maPile);

    return 0;
}
```

Scanned with CamScanner



Files

Les files ressemblent assez aux piles, si ce n'est qu'elles fonctionnent dans le sens inverse !

Scanned with CamScanner

Fonctionnement des files

- Dans ce système, les éléments s'entassent les uns à la suite des autres.
- Le premier qu'on fait sortir de la file est le premier à être arrivé.
- On parle ici d'algorithme **FIFO** (First In First Out), c'est-à-dire « Le premier qui arrive est le premier à sortir ».

Scanned with CamScanner

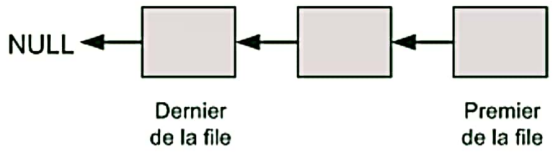
Fonctionnement des files

- En programmation, les files sont utiles pour mettre en attente des informations dans l'ordre dans lequel elles sont arrivées.
- Par exemple, dans un logiciel de *chat* (type messagerie instantanée), si vous recevez trois messages à peu de temps d'intervalle, vous les enfilez les uns à la suite des autres en mémoire.
- Vous vous occupez alors du premier message arrivé pour l'afficher à l'écran, puis vous passez au second, et ainsi de suite.

Scanned with CamScanner

Fonctionnement des files

- En C, une file est une liste chaînée où chaque élément pointe vers le suivant, tout comme les piles.
- Le dernier élément de la file pointe vers NULL.



Scanned with CamScanner

Création d'un système de file

- Nous allons créer une structure Element et une structure de contrôle File.
- Comme pour les piles, chaque élément de la file sera de type Element.

```
typedef struct Element Element;
struct Element
{
    int nombre;
    Element *suivant;
};
```

Scanned with CamScanner

Création d'un système de file

- N l'aide du pointeur premier, nous disposerons toujours du premier élément et nous pourrons remonter jusqu'au dernier.

```
typedef struct File File;
struct File
{
    Element *premier;
};
```

Scanned with CamScanner

Enfilage

- La fonction qui ajoute un élément à la file est appelée fonction « d'enfilage ».
- Il y a deux cas à gérer :
  - soit la file est vide, dans ce cas on doit juste créer la file en faisant pointer premier vers le nouvel élément créé ;
  - soit la file n'est pas vide, dans ce cas il faut parcourir toute la file en partant du premier élément jusqu'à arriver au dernier. On rajoutera notre nouvel élément après le dernier.

Scanned with CamScanner

```c
void enfiler(File *file, int nvNombre)
{
    Element *nouveau = malloc(sizeof(*nouveau));
    if (file == NULL || nouveau == NULL)
    {
        exit(EXIT_FAILURE);
    }

    nouveau->nombre = nvNombre;
    nouveau->suivant = NULL;

    if (file->premier != NULL) /* La file n'est pas vide */
    {
        /* On se positionne à la fin de la file */
        Element *elementActuel = file->premier;
        while (elementActuel->suivant != NULL)
        {
            elementActuel = elementActuel->suivant;
        }
        elementActuel->suivant = nouveau;
    }
    else /* La file est vide, notre élément est le premier */
    {
        file->premier = nouveau;
    }
}
```

Scanned with CamScanner

Enfilage

- La différence par rapport aux piles, est qu'il faut se placer à la fin de la file pour ajouter le nouvel élément.

Scanned with CamScanner

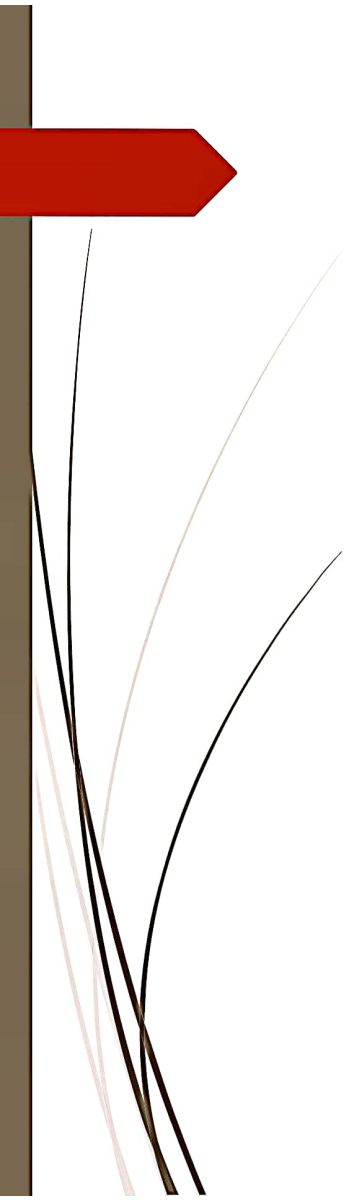
Défilage

- Le défilage ressemble étrangement au dépilage.
- Étant donné qu'on possède un pointeur vers le premier élément de la file, il nous suffit de l'enlever et de renvoyer sa valeur.

Scanned with CamScanner

```c
int defiler(File *file)
{
    if (file == NULL)
    {
        exit(EXIT_FAILURE);
    }

    int nombreDefile = 0;

    /* On vérifie s'il y a quelque chose à défiler */
    if (file->premier != NULL)
    {
        Element *elementDefile = file->premier;

        nombreDefile = elementDefile->nombre;
        file->premier = elementDefile->suivant;
        free(elementDefile);
    }

    return nombreDefile;
}
```

Scanned with CamScanner



À vous de jouer !

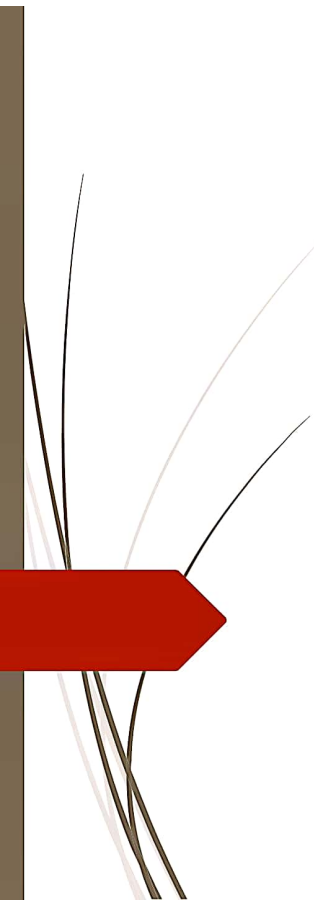
- Il resterait à écrire une fonction afficherFile.
- Puis le main

Scanned with CamScanner

En résumé

- Les piles et les files permettent d'organiser en mémoire des données qui arrivent au fur et à mesure.
- Elles utilisent un système de liste chaînée pour assembler les éléments.
- Dans le cas des piles,
  - les données s'ajoutent les unes au-dessus des autres.
  - Lorsqu'on extrait une donnée, on récupère la dernière qui vient d'être ajoutée (la plus récente).
  - On parle d'algorithme LIFO (*Last In First Out*).
- Dans les cas des files,
  - les données s'ajoutent les unes à la suite des autres.
  - On extrait la première donnée à avoir été ajoutée dans la file (la plus ancienne).
  - On parle d'algorithme FIFO (*First In First Out*).

Scanned with CamScanner



Arbres

Pr. Karima AISSAOUI

Karima.aissaoui1@usmba.ac.ma

Scanned with CamScanner

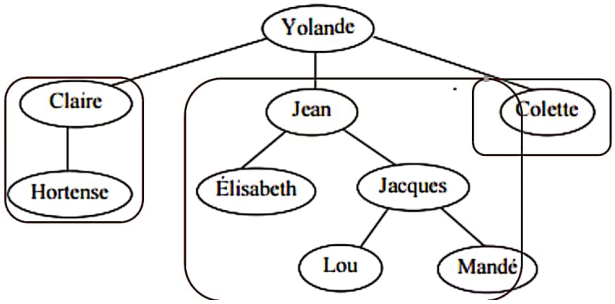
Arbres généraux

- Un arbre est une structure composée d'éléments appelés nœuds ; il est constitué d'un nœud particulier appelé racine et d'une suite ordonnée éventuellement vide A1, A2, …, Ap d'arbres disjoints appelés sous-arbres de la racine. Un arbre contient donc au moins un nœud : sa racine.

Scanned with CamScanner

Exemple

- la figure suivante représente l'arbre d'une descendance ; la racine de cet arbre contient Yolande ; il est constitué de trois sous-arbres (p = 3) :
  - A1, de racine Claire, contient Claire et Hortense ;
  - A2, de racine Jean, contient Jean, Élisabeth, Jacques, Lou et Mandé ;
  - A3, de racine Colette, est réduit à Colette.
  - L'arbre est ici dessiné de haut en bas (la racine est en haut) et les sous-arbres de gauche à droite ;



Scanned with CamScanner

Définitions

- Plusieurs définitions sont à retenir ; nous illustrons les définitions avec l'exemple ci-dessus en identifiant un nœud avec la chaîne de caractères qu'il contient :
  - les fils d'un nœud sont les racines de ses sous-arbres ; sur l'exemple, les fils de Jean sont Élisabeth et Jacques ;
  - le père d'un nœud x autre que la racine est l'unique nœud dont x est un fils ; sur l'exemple, Jacques est le père de Lou et de Mandé ;
  - la racine d'un arbre n'a pas de père ;
  - un nœud interne est un nœud qui a au moins un fils ; sur l'exemple, Claire est un nœud interne ;
  - une feuille d'un arbre est un nœud sans fils ; sur l'exemple, Mandé est une feuille ;
  - les ancêtres d'un nœud « a » sont les nœuds qui sont sur le chemin entre la racine (incluse) et « a » (inclus) ; les ancêtres de « a » différents de « a » sont les ancêtres propres de « a » ; sur l'exemple, les ancêtres de Jacques sont Jacques, Jean et Yolande ;

Scanned with CamScanner

Définitions

- les descendants d'un nœud « a » sont les nœuds qui appartiennent au sous-arbre de racine « a » ; les descendants de « a » différents de « a » sont les descendants propres de « a » ; sur l'exemple, les descendants de Jean sont Jean, Élisabeth, Jacques, Lou et Mandé.
- la profondeur d'un nœud est la longueur du chemin allant de la racine à ce nœud ;
- la profondeur de la racine est donc nulle et la profondeur d'un nœud autre que la racine vaut un de plus que la profondeur de son père ;
- la profondeur d'un nœud peut aussi être appelée niveau ou hauteur du nœud ; sur l'exemple, la profondeur d'Élisabeth vaut 2 ;
- la hauteur d'un arbre est la profondeur maximum de ses nœuds ; la hauteur de l'arbre donné en exemple vaut 3.

Scanned with CamScanner

- Pour connaître un arbre, on procède généralement en mémorisant la racine (ou plus précisément son adresse) ; il suffit alors que chaque nœud connaisse ses nœuds fils pour pouvoir retrouver tout l'arbre.

Scanned with CamScanner

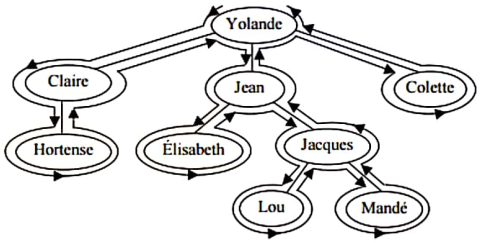
Parcours d'un arbre général

- Parcourir un arbre, c'est examiner une fois et une seule chacun de ses nœuds, l'ordre des examens dépendant d'une certaine règle.
- On distingue les deux types suivants.

Scanned with CamScanner

Parcours en préordre

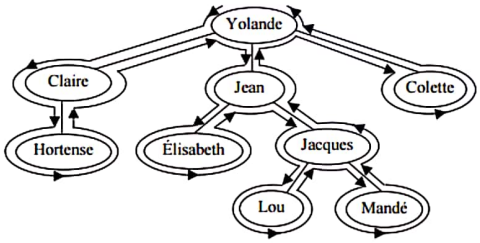
- Parcourir un arbre en préordre, c'est :
  - Examiner la racine ;
  - Parcourir en préordre le premier sous-arbre ;
  - …
  - parcourir en préordre le dernier sous-arbre.
- Exemple : le parcours en préordre de l'arbre donné en exemple considère successivement les nœuds Yolande, Claire, Hortense, Jean, Élisabeth, Jacques, Lou, Mandé, Colette.



Scanned with CamScanner

Parcours en postordre

- Parcourir un arbre en postordre, c'est :
  - parcourir en postordre le premier sous-arbre ;
  - …
  - parcourir en postordre le dernier sous-arbre ;
  - examiner la racine.
- Exemple : le parcours en postordre de l'arbre donné en exemple considère successivement les nœuds Hortense, Claire, Élisabeth, Lou, Mandé, Jacques, Jean, Colette, Yolande.
- On part de la racine et on suit le tracé de l'arbre comme l'indique la figure ci-dessous. On s'aperçoit que l'examen en préordre est obtenu en examinant les nœuds à la première rencontre alors que l'examen en postordre est obtenu en examinant les nœuds à la dernière rencontre.



Scanned with CamScanner

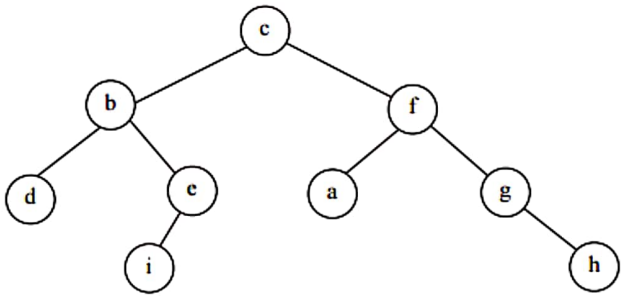
Arbres binaires

- Un arbre binaire est soit vide, soit constitué d'un nœud particulier appelé racine et de deux sous-arbres binaires disjoints appelés sous-arbre gauche et sous-arbre droit.

Scanned with CamScanner

Exemple d'arbre binaire

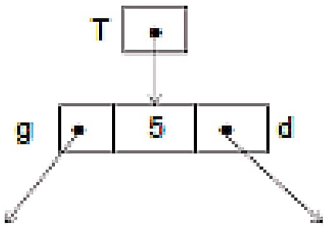
- La terminologie est la même que celle des arbres généraux ; néanmoins, si on considère les fils d'un nœud, on parlera, s'ils existent, du fils gauche et du fils droit.
- Sur l'exemple, on remarque que certains nœuds peuvent avoir seulement un fils droit ou seulement un fils gauche ;
- Par exemple, le nœud « e » (i.e. contenant la donnée e) admet uniquement un fils gauche, le nœud « i », et pas de fils droit ;



Scanned with CamScanner

Arbres binaires

- Pour coder un arbre binaire, on fait souvent correspondre à chaque nœud **une structure** contenant **la donnée** et **deux adresses**, une adresse pour chacun des deux nœuds fils.
- Il suffit alors de mémoriser l'adresse de la racine pour pouvoir reconstituer tout l'arbre.
- Une adresse nulle indique un arbre binaire vide.



Scanned with CamScanner

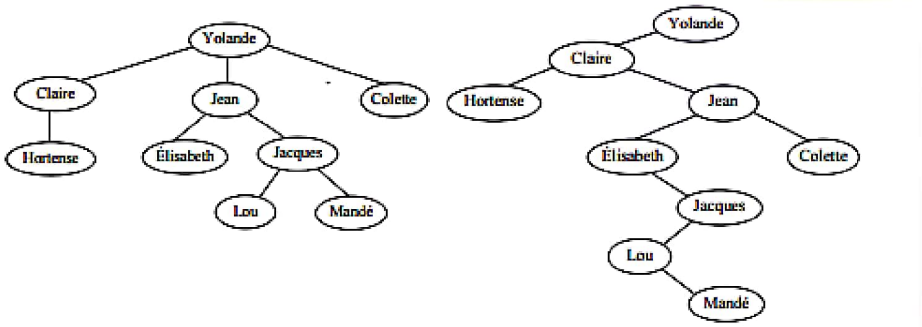
Remarque

- On code souvent un arbre général en utilisant un arbre binaire ;
- On associe à chaque nœud de l'arbre initial un nœud de l'arbre binaire ;
- Le fils gauche d'un nœud dans l'arbre binaire correspond au premier fils du nœud correspondant de l'arbre initial (on met une adresse nulle s'il n'y a aucun fils)
- Alors que le fils droit de ce nœud correspond au premier « frère cadet », c'est-à-dire au nœud de l'arbre initial qui a même père que le nœud considéré et qui se trouve immédiatement à sa droite (on met une adresse nulle s'il n'y a aucun frère cadet).

Scanned with CamScanner

Remarque

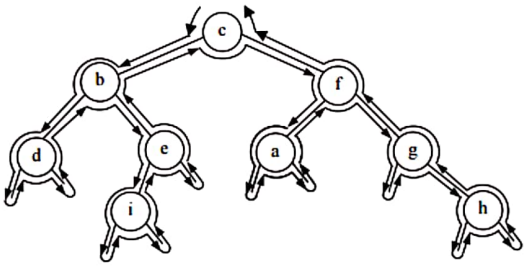
- Ainsi, l'arbre général donné en exemple précédemment et retracé ci-dessous à gauche peut être codé avec l'arbre binaire représenté à droite, en changeant bien entendu l'interprétation des liens.



Scanned with CamScanner

Parcours(Le parcours en ordre symétrique)

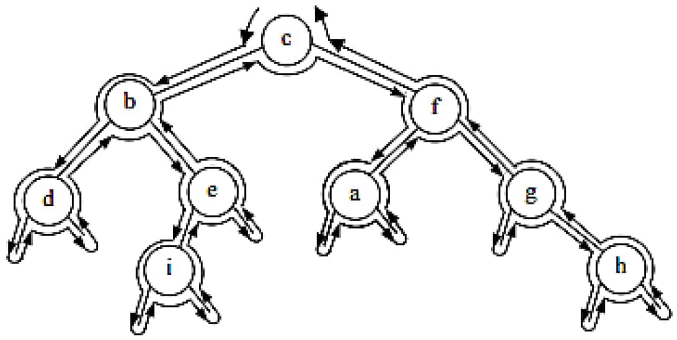
- Les deux parcours définis pour les arbres généraux sont aussi définis pour les arbres binaires.
- Pour l'exemple ci-dessus, le parcours en préordre donne l'ordre : c, b, d, e, i, f, a, g, h et le parcours en postordre donne l'ordre : d, i, e, b, a, h, g, f, c.
- On considère pour les arbres binaires un troisième ordre : l'ordre symétrique. Parcourir un arbre binaire en ordre symétrique, c'est : si l'arbre binaire est non vide, alors
  - parcourir en ordre symétrique le sous-arbre gauche ;
  - examiner la racine ;
  - parcourir en ordre symétrique le sous-arbre droit.



Scanned with CamScanner

Parcours(Le parcours en ordre symétrique)

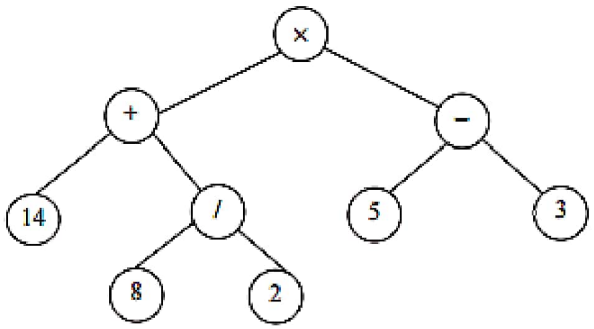
- Le parcours en ordre symétrique de l'arbre binaire donné en exemple conduit à l'ordre : d, b, i, e , c, a, f, g, h.



Scanned with CamScanner

Exemple d'application

- On peut utiliser un arbre binaire pour représenter une expression arithmétique composée à partir des entiers et des quatre opérateurs binaires : +, –, ×, /.
- L'arbre ci-dessous représente l'expression : (14 + (8 / 2)) × (5 – 3)



Scanned with CamScanner

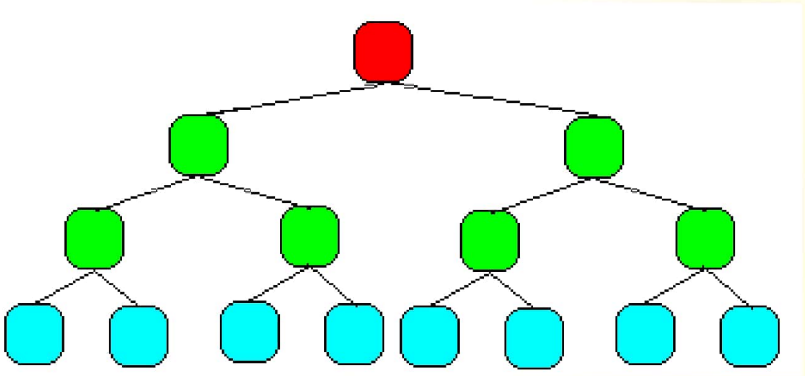
Exemple d'application

- Par construction, dans cet arbre, tout nœud interne a un fils gauche et un fils droit.
- Les trois parcours de cet arbre donnent :
  - Parcours en préordre : × + 14 / 8 2 – 5 3 ; La forme polonaise de l'expression donne 14 + 8 / 2 × 5 – 3 ;
  - parcours en ordre infixe : 14 + 8 / 2 × 5 – 3 ; il faudrait ajouter les parenthèses pour retrouver l'expression initiale ;
  - parcours en postordre : 14 8 2 / + 5 3 – × ; La forme polonaise inverse de l'expression donne 14 + 8 / 2 × 5 – 3 .

Scanned with CamScanner

Quelques arbres binaires particuliers

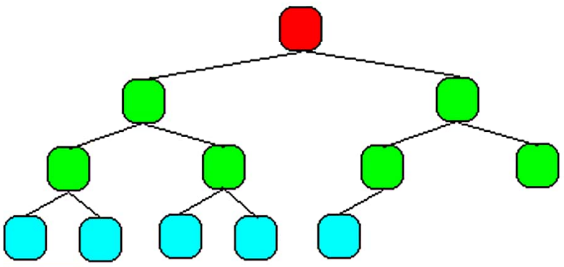
- Un arbre binaire est dit complet si tout nœud interne a exactement deux fils.



Scanned with CamScanner

Quelques arbres binaires particuliers

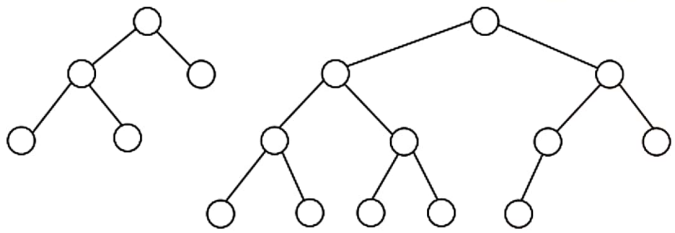
- Un arbre binaire est dit parfait si, en appelant h la hauteur de l'arbre, les niveaux de profondeur 0, 1, …, h – 1 sont complètement remplis alors que le niveau de profondeur h est rempli en partant de la gauche ;
- Un arbre est parfait si tout ses niveaux sont remplis sauf éventuellement le dernier niveau qui a toutes ses feuilles ramenées complètement à gauche.



Scanned with CamScanner

Quelques arbres binaires particuliers

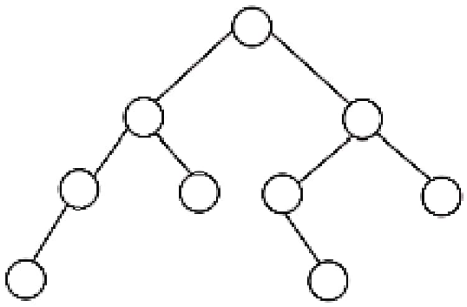
- la structure d'un arbre binaire parfait est complètement déterminée par son nombre de nœuds ; nous donnons ci-après les arbres binaires parfaits à 5 nœuds et à 12 nœuds.



Scanned with CamScanner

Quelques arbres binaires particuliers

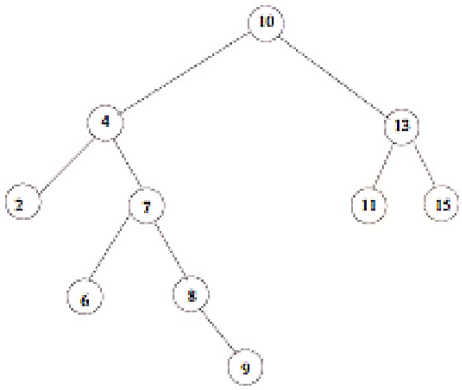
- En revanche, l'arbre ci-contre n'est pas parfait.



Scanned with CamScanner

Arbres binaires de recherche

- Un arbre binaire de recherche est un arbre binaire qui possède la propriété fondamentale suivante:
  - Tous les nœuds du sous-arbre de gauche d'un nœud de l'arbre ont une valeur inferieure ou égale a la sienne.
  - Tous les nœuds du sous-arbre de droite d'un nœud de l'arbre ont une valeur supérieure ou égale a la sienne.



Scanned with CamScanner

Recherche dans l'arbre

- Un arbre binaire de recherche est fait pour faciliter la recherche d'informations.
- La recherche d'un nœud particulier de l'arbre peut être définie simplement de manière récursive:

  Soit un sous-arbre de racine « n_i »,

  - si la valeur recherchée est celle de la racine « n_i », alors la recherche est terminée. On a trouvé le nœud recherché.
  - sinon,
    - si « n_i » est une feuille (pas de fils) alors la recherche est infructueuse et l'algorithme se termine.
    - si la valeur recherchée est plus grande que celle de la racine alors on explore le sous-arbre de droite c'est à dire que l'on remplace « n_i » par son nœud fils de droite et que l'on relance la procédure de recherche à partir de cette nouvelle racine.
    - De la même manière, si la valeur recherchée est plus petite que la valeur de « n_i », on remplace « n_i » par son nœud fils de gauche avant de relancer la procédure..

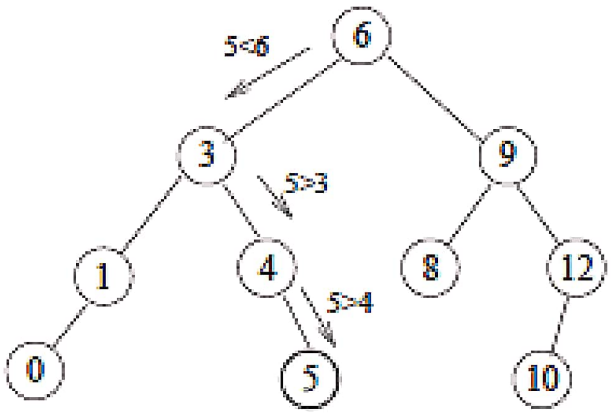
Scanned with CamScanner

Ajout d'un élément

- Pour conserver les propriétés d'un arbre binaire de recherche nécessite de, l'ajout d'un nouvel élément ne peut pas se faire n'importe comment.
- L'algorithme récursif d'ajout d'un élément peut s'exprimer ainsi:
  - Soit v la valeur de l'élément à insérer.
  - soit x la valeur du nœud racine « n_i » d'un sous-arbre.
  - Si « n_i » n'existe pas, le créer avec la valeur v. fin.
    - sinon
      - si v est plus grand que x,
        - remplacer « n_i » par son fils droit.
        - recommencer l'algorithme à partir de la nouvelle racine.
      - sinon
        - remplacer « n_i » par son fils gauche.
        - recommencer l'algorithme à partir de la nouvelle racine.

Scanned with CamScanner

Ajout d'un élément



Scanned with CamScanner

Implémentation

- En langage C, un nœud d'un arbre binaire peut être représente par une structure contenant un champ donnée et deux pointeurs vers les nœuds fils:

```c
struct s_arbre
{
    int valeur;
    struct s_arbre * gauche;
    struct s_arbre * droit;
};
typedef struct s_arbre t_arbre;
```

Scanned with CamScanner

Implémentation

- La fonction d'insertion qui permet d'ajouter un élément dans l'arbre et donc de le créer de manière à ce qu'il respecte les propriétés d'un arbre binaire de recherche peut s'écrire ainsi:

```
void insertion(t_arbre ** noeud, int v)
{
 if (*noeud==NULL) /* si le noeud n'existe pas, on le cree */
{
     *noeud=(t_arbre*) malloc(sizeof(t_arbre));
     (*noeud)->valeur=v;
     (*noeud)->gauche=NULL;
     (*noeud)->droit=NULL;
}
 else
{
     if (v>(*noeud)->valeur)
         insertion(&(*noeud)->droit,v); /* aller a droite */
     else
         insertion(&(*noeud)->gauche,v); /* aller a gauche */
}
}
```

Scanned with CamScanner

Suppression

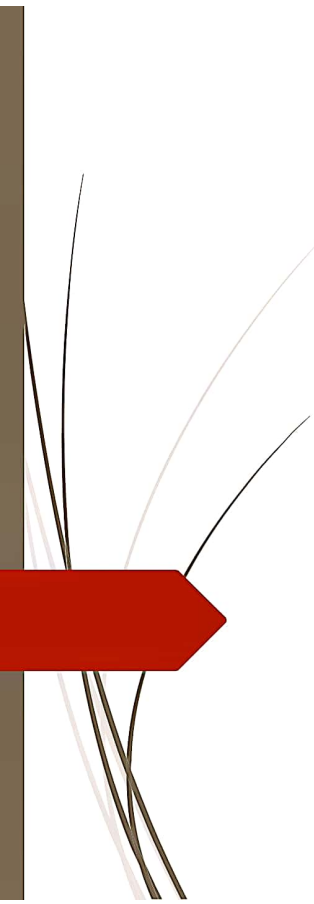
- L'algorithme permettant de supprimer un nœud d'un arbre est le suivant:
  - Si le nœud à supprimer n'a pas de fils ➔ il suffit de le supprimer.
  - Si le nœud à supprimer a un seul fils ➔ On le remplace par son fils et on supprime ce dernier.
  - Si le nœud à supprimer a deux fils ➔ On le remplace par l'extrémité du bord gauche du sous-arbre droit.

Scanned with CamScanner

Les opérations sur les arbres

- Les opérations sur les arbres
  - Ajouter un élément
  - Déterminer si un élément existe dans l'arbre
  - Supprimer un élément de l'arbre
  - Retourner la hauteur
  - Retourner le nombre de nœud
  - ....

Scanned with CamScanner



LES TABLES DE HACHAGE

Pr. Karima AISSAOUI
Karima.aissaoui1@usmba.ac.ma

Scanned with CamScanner

Introduction

- Les listes chaînées ont un gros défaut lorsqu'on souhaite lire ce qu'elles contiennent : il n'est pas possible d'accéder directement à un élément précis. Il faut parcourir la liste en avançant d'élément en élément jusqu'à trouver celui qu'on recherche.

➔ Cela pose des problèmes de performance dès que la liste chaînée devient volumineuse. « Imaginez une liste chaînée de 1 000 éléments où celui que l'on recherche est tout à la fin ! »

- Les tables de hachage représentent une autre façon de stocker des données.

➔ Elles sont basées sur les tableaux du langage C.

➔ Leur gros avantage : Elles permettent de retrouver instantanément un élément précis, que la table contient 100, 1 000, 10 000 cases ou plus encore !

Scanned with CamScanner

Pourquoi utiliser une table de hachage ?

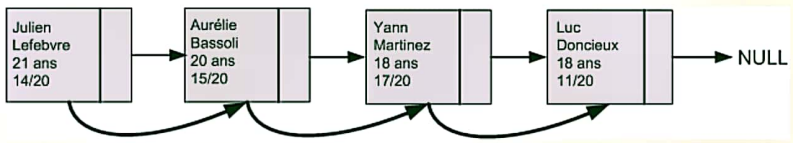
Les listes chaînées sont particulièrement souples, nous avons pu le constater : il est possible d'ajouter ou de supprimer des cases à tout moment, alors qu'un tableau est « figé » une fois qu'il a été créé.

Toutefois, les listes chaînées ont quand même un gros défaut : si on cherche à récupérer un élément précis de la liste, il faut parcourir celle-ci en entier jusqu'à ce qu'on le retrouve !

Scanned with CamScanner

Pourquoi utiliser une table de hachage ?

- Prenons une liste chaînée qui contient des informations sur des élèves :
  - leur nom,
  - leur âge et
  - leur moyenne.
- Chaque élève sera représenté par une structure « Eleve »
- Si je veux retrouver les informations sur Luc Doncieux dans la fig. suivante, il va falloir parcourir toute la liste pour se rendre compte qu'il était à la fin !



Scanned with CamScanner

Pourquoi utiliser une table de hachage ?

- Dans cet exemple, notre liste chaînée ne contient que quatre éléments. Mais imaginez maintenant que celui-ci se trouve à la fin d'une liste chaînée contenant 10 000 éléments ! Ce n'est pas acceptable de devoir parcourir jusqu'à 10 000 éléments pour retrouver une information. C'est là que les tables de hachage entrent en jeu.

Scanned with CamScanner

Qu'est-ce qu'une table de hachage ?

- Les tableaux ne connaissaient pas ce problème. Ainsi, pour accéder à l'élément d'indice 2 dans mon tableau, il me suffisait d'écrire ceci :

```
int tableau[4] = {12, 7, 14, 33};
printf("%d", tableau[2]);
```

- *Si on donne tableau[2], on va directement à la case mémoire où se trouve stocké le nombre 14. Il ne parcourt pas les cases du tableau une à une.*
- *Mais avec les tableaux, on perd l'avantage des listes chaînées qui nous permettaient d'ajouter et de retirer des cases à tout moment !*

Scanned with CamScanner

Qu'est-ce qu'une table de hachage ?

- Donc, les listes chaînées sont plus flexibles. Les tableaux, eux, permettent un accès plus rapide.

➔ Les **tables de hachage** constituent quelque part un compromis entre les deux.

- Mais, Il y a un défaut important avec les tableaux : les cases sont identifiées par des numéros qu'on appelle des **indices**.

➔ Il n'est pas possible de demander à l'ordinateur les données qui se trouvent à la case "Luc Doncieux" .

➔ Pour retrouver l'âge et la moyenne de Luc Doncieux, on ne peut donc pas écrire :

```
tableau["Luc Doncieux"];
```

Scanned with CamScanner

Qu'est-ce qu'une table de hachage ?

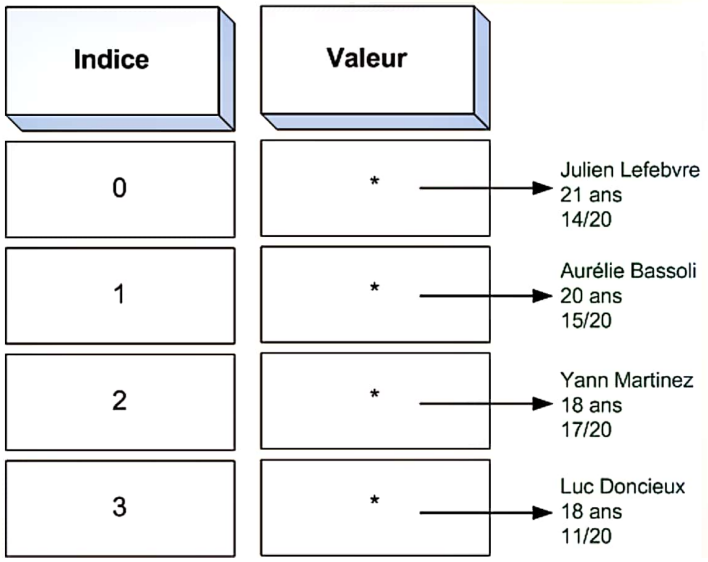
- Ce serait pourtant pratique de pouvoir accéder à une case du tableau rien qu'avec le nom ! Eh bien avec les tables de hachage, c'est possible.
- ➔ Les tables de hachage ne font pas « partie » du langage C. Il s'agit d'un concept.

- ➔ *Puisque notre tableau doit forcément être numéroté par des indices, comment fait-on pour retrouver le bon numéro de case si on connaît seulement le nom « Luc Doncieux » ?*

Scanned with CamScanner

Qu'est-ce qu'une table de hachage ?

- En effet, un tableau reste un tableau et celui-ci ne fonctionne qu'avec des indices numérotés. Imaginez un tableau correspondant à la fig. suivante : chaque case a un indice et possède un pointeur vers une structure de type Eleve.



Scanned with CamScanner

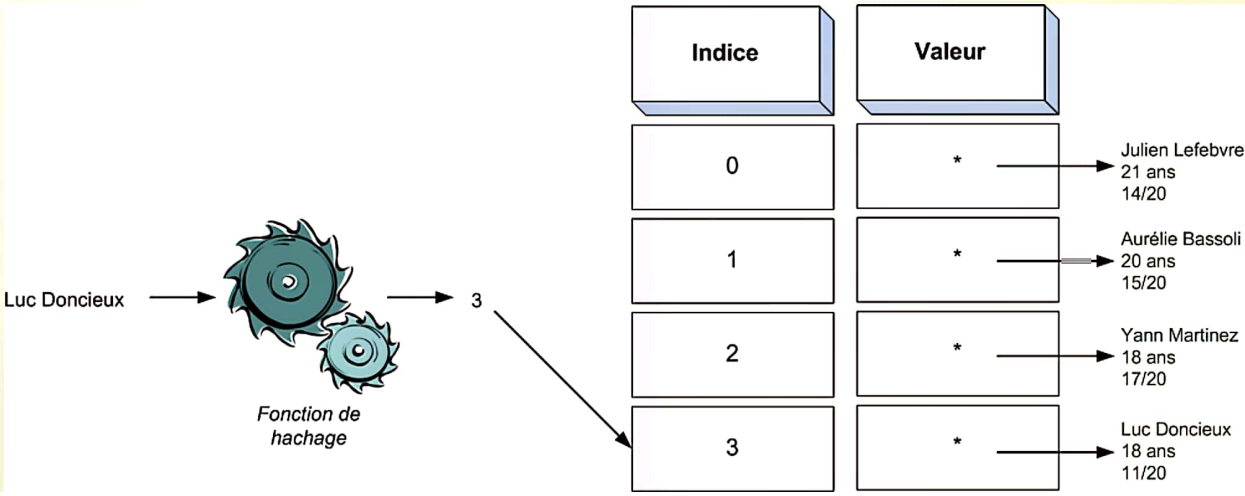
Qu'est-ce qu'une table de hachage ?

- Si on veut retrouver la case correspondant à Luc Doncieux, il faut pouvoir transformer son nom en indice du tableau. Ainsi, il faut pouvoir faire l'association entre chaque nom et un numéro de case de tableau :
  - Julien Lefebvre = 0 ;
  - Aurélie Bassoli = 1 ;
  - Yann Martinez = 2 ;
  - Luc Doncieux = 3.
- Mais On ne peut pas écrire tableau["Luc Doncieux"]. Ce n'est pas valide en C.

Scanned with CamScanner

Qu'est-ce qu'une table de hachage ?

- La question est : comment transformer une chaîne de caractères en numéro ?
- Il faut écrire une fonction qui prend en entrée une chaîne de caractères, fait des calculs avec, puis retourne en sortie un numéro correspondant à cette chaîne. Ce numéro sera l'indice de la case dans notre tableau.



Scanned with CamScanner

Écrire une fonction de hachage

- Toute la difficulté consiste à écrire une fonction de hachage correcte.

➔Comment transformer une chaîne de caractères en un nombre unique ?

Prenons donc un tableau de 100 cases dans lequel on va stocker des pointeurs vers des structures Eleve.

```
Eleve* tableau[100];
```

Scanned with CamScanner

Écrire une fonction de hachage

- Nous devons écrire une fonction qui, à partir d'un nom, génère un nombre compris entre 0 et 99 (les indices du tableau).
- Il existe des méthodes mathématiques pour « hacher » des données, c'est-à-dire les transformer en nombres.
- Les algorithmes MD5 et SHA1 sont des fonctions de hachage célèbres.
- Il est possible d'inventer une fonction de hachage. Ici, pour faire simple, on propose d'additionner les valeurs ASCII de chaque lettre du nom, c'est-à-dire pour Luc Doncieux faire la somme suivante :
- 'L' + 'u' + 'c' + ' ' + 'D' + 'o' + 'n' + 'c' + 'i' + 'e' + 'u' + 'x'

Scanned with CamScanner

Écrire une fonction de hachage

- On va toutefois avoir un problème : cette somme dépasse 100 ! Comme notre tableau ne fait que 100 cases, si on s'en tient à ça, on risque de sortir des limites du tableau.
- Chaque lettre dans la table ASCII peut être numérotée jusqu'à 255. On a donc vite fait de dépasser 100.

➔ Pour régler le problème, on peut utiliser l'opérateur modulo %.

`sommeLettres % 100`

… on obtiendra forcément un nombre compris entre 0 et 99. Par exemple, si la somme fait 4315, le reste de la division par 100 est 15. La fonction de hachage retournera donc 15.

Scanned with CamScanner

Écrire une fonction de hachage

- Voici à quoi pourrait ressembler cette fonction :

```
int hachage(char *chaine)
{
    int i = 0, nombreHache = 0;
    for (i = 0 ; chaine[i] != '\0' ; i++)
    {
        nombreHache += chaine[i];
    }
    nombreHache %= 100;
    return nombreHache;
}
```

Scanned with CamScanner

Écrire une fonction de hachage

- Si on lui envoie hachage("Luc Doncieux"), elle renvoie 55.
- Avec hachage("Yann Martinez"), on obtient 80.

➔ Grâce à cette fonction de hachage, vous savez donc dans quelle case de votre tableau vous devez placer vos données !
➔ Lorsque vous voudrez y accéder plus tard pour en récupérer les données, il suffira de hacher à nouveau le nom de la personne pour retrouver l'indice de la case du tableau où sont stockées les informations !

Il est recommandé de créer une fonction de recherche qui se chargera de hacher la clé (le nom) et de renvoyer un pointeur vers les données recherchée. Cela donnerait par exemple :
infosSurLuc = rechercheTableHachage(tableau, "Luc Doncieux");

Scanned with CamScanner

Gérer les collisions

- Quand la fonction de hachage renvoie le même nombre pour deux clés différentes, on dit qu'il y a **collision**.
- Par exemple dans notre cas, si nous avions un anagramme de Luc Doncieux qui s'appelle Luc*Doncueix*, la somme des lettres est la même, donc le résultat de la fonction de hachage sera le même !
- Deux raisons peuvent expliquer une collision.
  - La fonction de hachage n'est pas très performante. C'est notre cas. Nous avons écrit une fonction très simple (mais néanmoins suffisante) pour nos exemples.

  ➔ Les fonctions MD5 et SHA1 mentionnées plus tôt sont de bonne qualité car elles produisent très peu de collisions. Notez que SHA1 est aujourd'hui préférée à MD5 car c'est celle des deux qui en produit le moins.
  - Le tableau dans lequel on stocke nos données est trop petit. Si on crée un tableau de 4 cases et qu'on souhaite stocker 5 personnes, on aura à coup sûr une collision, c'est-à-dire que notre fonction de hachage donnera le même indice pour deux noms différents.
- Si une collision survient, pas de panique ! Deux solutions s'offrent à vous au choix : l'adressage ouvert et le chaînage.

Scanned with CamScanner

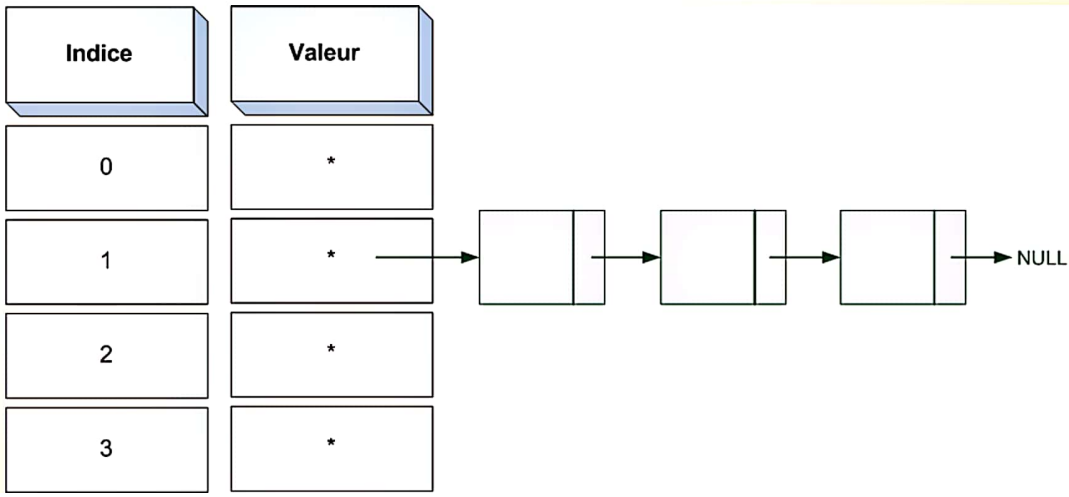
L'adressage ouvert

- S'il reste de la place dans votre tableau, vous pouvez utiliser la technique dite du **hachage linéaire**. Le principe est simple. La case est occupée ? Pas de problème, allez à la case suivante. Ah, elle est occupée aussi ? Allez à la suivante !
- Ainsi de suite, continuez jusqu'à trouver la prochaine case libre dans le tableau. Si vous arrivez à la fin du tableau, retournez à la première case et continuez.
- Cette méthode est très simple à mettre en place, mais si vous avez beaucoup de collisions, vous allez passer beaucoup de temps à chercher la prochaine case libre.
- Il existe des variantes (hachage double, hachage quadratique…) qui consistent à hacher à nouveau selon une autre fonction en cas de collision. Elles sont plus efficaces mais plus complexes à mettre en place.

Scanned with CamScanner

Le chaînage

- Une autre solution consiste à créer une **liste chaînée** à l'emplacement de la collision. Vous avez deux données (ou plus) à stocker dans la même case ? Utilisez une liste chaînée et créez un pointeur vers cette liste depuis le tableau.



Scanned with CamScanner

- On en revient au défaut des listes chaînées : s'il y a 300 éléments à cet emplacement du tableau, il va falloir parcourir la liste chaînée jusqu'à trouver le bon.
- Les listes chaînées ne sont pas toujours idéales, mais les tables de hachage ont aussi leurs limites. On peut combiner les deux pour tenter de tirer le meilleur de chacune de ces structures de données.

Scanned with CamScanner

En résumé

- Les listes chaînées sont flexibles, mais il peut être long de retrouver un élément précis à l'intérieur car il faut les parcourir case par case.
- Les tables de hachage sont des tableaux. On y stocke des données à un emplacement déterminé par une fonction de hachage.
- La fonction de hachage prend en entrée une clé (ex. : une chaîne de caractères) et retourne en sortie un nombre.
- Ce nombre est utilisé pour déterminer à quel indice du tableau sont stockées les données.
- Une bonne fonction de hachage doit produire peu de collisions, c'est-à-dire qu'elle doit éviter de renvoyer le même nombre pour deux clés différentes.
- En cas de collision, on peut utiliser l'adressage ouvert (recherche d'une autre case libre dans le tableau) ou bien le chaînage (combinaison avec une liste chaînée).

Scanned with CamScanner

Algorithmes de tri

Pr. Karima AISSAOUI

Karima.aissaoui1@usmba.ac.ma

Scanned with CamScanner

Tri

- Trier un ensemble d'objets consiste à ordonner ces objets en fonction de clés et d'une relation d'ordre définie sur cette clé.
- Par exemple, chaque étudiant inscrit à l'ENSAO reçoit un numéro qui constitue la clé d'accès à son dossier.
- On trie des objets lorsqu'ils sont nombreux et qu'on doit pouvoir y accéder rapidement.

Scanned with CamScanner

Tri

- On dit qu'un algorithme de tri est
  - en place, s'il n'utilise pas d'espace mémoire supplémentaire pour stocker les objets,
  - par comparaison, si le tri s'effectue en comparant les objets entres eux. Il existe des tris qui ne comparent pas les objets entre eux : lorsque les clés sont peu nombreuses, connues à l'avance, et peuvent être indexées facilement.
  - Un tri est stable s'il préserve l'ordre d'apparition des objets en cas d'égalité des clés. Cette propriété est utile par exemple lorsqu'on trie successivement sur plusieurs clés différentes. Si l'on veut ordonner les étudiants par rapport à leur nom puis à leur moyenne générale, on veut que les étudiants qui ont la même moyenne apparaissent dans l'ordre lexicographique de leurs noms.

Scanned with CamScanner

Algorithmes de tri

Dans ce cours nous distinguerons les tris suivants:

- tri à bulle,
- tri par insertion,
- tri par sélection

- En général les objets à trier sont stockés dans des tableaux indexés, mais ce n'est pas toujours le cas. Lorsque les objets sont stockés dans des listes chainées, on peut par exemple les recopier dans un tableau temporaire.

Scanned with CamScanner

Tri par sélection

- Le tri par sélection consiste simplement à sélectionner l'élément le plus petit de la suite à trier, à l'enlever, et à répéter itérativement le processus tant qu'il reste des éléments dans la suite.
- Au fur et à mesure les éléments enlevés sont stockés dans une pile.
- Lorsque la suite à trier est stockée dans un tableau on s'arrange pour représenter la pile dans le même tableau que la suite : la pile est représentée au début du tableau, et chaque fois qu'un élément est enlevé de la suite il est remplacé par le premier élément qui apparaît à la suite de la pile, et prends sa place.
- Lorsque le processus s'arrête la pile contient tous les éléments de la suite triés dans l'ordre croissant.

Scanned with CamScanner

Tri par insertion

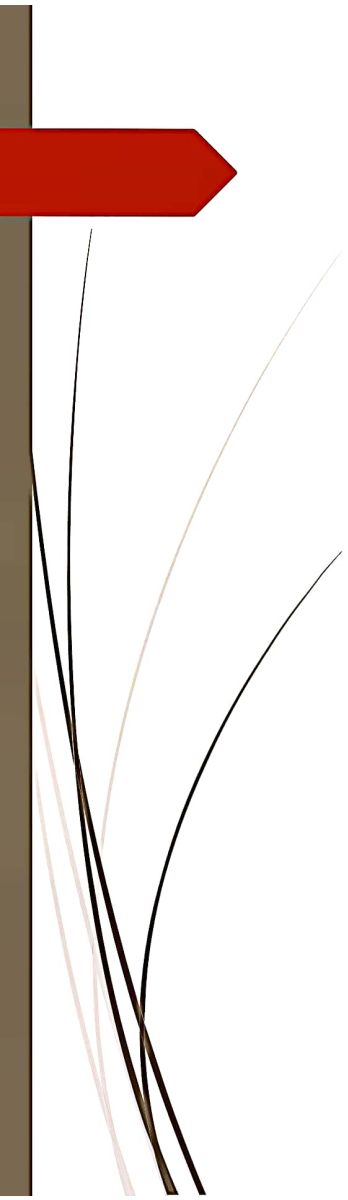
- Le tri par insertion consiste à insérer les éléments de la suite les uns après les autres dans une suite triée initialement vide.
- Lorsque la suite est stockée dans un tableau la suite triée en construction est stockée au début du tableau.
- Lorsque la suite est représentée par une liste chainée on insère les maillons les uns après les autres dans une nouvelle liste initialement vide.

Scanned with CamScanner

Tri à bulle

- Le tri à bulle consiste à parcourir le tableau, tant qu'il n'est pas trié, et à permuter les couples d'éléments consécutifs mal ordonnés.
- On sait que le tableau est trié si lors d'un parcours, aucun couple d'éléments n'a été permuté.

Scanned with CamScanner



Encore plus …

- tris par fusion,
- tri par tas
- tri rapide

Scanned with CamScanner

Exercice

- Ecrire une méthode triInsert qui trie par insertion une liste chainée.

Scanned with CamScanner

LECTURE & ÉCRITURE DANS DES FICHIERS EN C

Pr. Karima AISSAOUI

Karima.aissaoui1@usmba.ac.ma

Scanned with CamScanner

Les fichiers

- **Un programme a en général besoin :**
  - De lire des données (texte, nombres, images, sons, mesures, …)
  - De sauvegarder des résultats (texte, nombres, images, sons, signaux générés, …)

Cela se fait en lisant et en écrivant dans des fichiers

- Pour manipuler un fichier, on utilise un **pointeur** sur une donnée spécifique dont le type est **FILE** (structure prédéfinie que nous n'avons pas besoin de connaître précisément) :

```
FILE *fichier
```

- La variable `fichier` contiendra l'adresse en mémoire du début du fichier

Scanned with CamScanner

Ouverture-fermeture de fichiers

• **Ouverture** d'un fichier à l'aide de la fonction **fopen :**

```
fichier = fopen("C:/Data/fichier1.txt","r");
```

Mode d'ouverture

Cette fonction renvoie un pointeur sur le fichier ouvert.

fopen est définie dans le fichier stdio.h par :

```
FILE *fopen (char *nom, char *mode)
```

•`nom` est une chaîne de caractères (tableau de caractères) contenant le nom du fichier ou bien un flot de données standard (stdin, …)

•`mode` désigne le type de traitement des données

- `"r"` (read) : lecture (si le fichier existe)
- `"w"` (write) : écriture (le fichier est écrasé s'il existe et s'il n'existe pas, il est crée)
- `"a"` (append) : écriture à la fin d'un fichier existant

Activer Win

Scanned with CamScanner

Ouverture-fermeture de fichiers

Où se trouve le fichier ouvert ?
→Dans le répertoire de travail (plus exactement là où est le fichier exécutable .exe)

Comment travailler sur un fichier situé ailleurs ?
→ Le chemin absolu d'accès au fichier peut être donné in extenso (attention les \ de windows deviennent des / en C)

```
fichier = fopen("C:/Data/fichier1.txt","r");
```

```
fichier = fopen("fichier1","r");
```

A quoi sert l'extension .txt ?
→A rien pour le Langage C.
→Elle permet en revanche à l'OS de l'ordinateur de sélectionner le programme permettant d'ouvrir le fichier.

Scanned with CamScanner

Ouverture-fermeture de fichiers

• **Fermeture** d'un fichier à l'aide de la fonction **fclose** :

Important : ***Il faut toujours fermer un fichier après l'avoir utilisé***
→ Afin de libérer la mémoire

```
fclose(fichier) ;
```

fclose est définie dans le fichier stdio.h par :

```
int fclose(FILE *fichier) ;
```

Scanned with CamScanner

Écriture-lecture de fichiers textes

• **Écriture :** `fprintf(FILE *fichier,char *proto,…);`

Exemple :
```
double a;
fprintf(fichier,«%lf»,a);
```

→ ***Presque même syntaxe que printf !***

```
printf("%lf",a);
```

• **Lecture :** `fscanf(FILE *fichier,char *proto,…);`

Exemple :
```
double a;
fscanf(fichier,"%lf",&a);
```

→ ***Presque même syntaxe que scanf !***

```
scanf("%lf",&a);
```

Scanned with CamScanner

Exemple: écriture dans un fichier

```
include <stdio.h>
void main()
{
    double a=1.5, b=2.5;
    FILE   *fichier;

    // Ouverture du fichier en écriture grâce à "w"
    fichier = fopen("essai.txt","w");

    // Verifier que le fichier a bien été ouvert
    if (fichier != NULL)
    {
        // Ecriture
        fprintf(fichier,"%lf\n",a);
        fprintf(fichier,"%lf\n",b);
        // Fermeture du fichier
        fclose(fichier);
    }
}
```

Scanned with CamScanner

Exemple: lecture à partir d'un fichier

```
#include <stdio.h>
void main()
{
	int	i;
	double	tab[2];
	FILE	*fichier;

	// Ouverture du fichier en lecture grâce à "r"
	fichier = fopen("essai.txt","r");
	if (fichier != NULL)
	{
		for(i=0;i<2;i++)
			fscanf(fichier,"%lf\n",tab+i);
		fclose(fichier);
	}
	for(i=0;i<2;i++) printf("%lf\n",tab[i]);
}
```

Scanned with CamScanner

Format d'un fichier

- Les fichiers précédents ont été écrits au format **« texte »**

  → On a utilisé jusqu'ici les fonctions : `fprintf` et `fscanf`

  → Tout se passe comme si l'affichage de la fenêtre console était envoyé dans le fichier.
  → Ou bien comme si le fichier se comportait comme les saisies clavier pour envoyer des données au programme.

  → Les caractères sont stockés avec le code ASCii : *American Standard Code for Information Interchange*

  → Avec un langage évolué tel que le C :
  → Nous n'avons pas à connaître le code ASCii
  → Nous n'avons pas à connaître la représentation des nombres dans les mémoires

Scanned with CamScanner

Format d'un fichier

• Lorsqu'on a des **données numériques**, il est souvent plus efficace de recopier directement le contenu de la mémoire donc de les écrire sous forme **« binaire ».**

- Soit un nombre `1.345643355454E-18`
  - Au format `double`, il occupe 8 octets
  - Au format `ASCII`, il occupe 20 octets (1 octet par caractère)

• On a 2 types de fichiers : fichiers **« texte»** et **« binaire ».**
• Le format binaire va occuper beaucoup moins de place en mémoire
• Comme il s'agit d'une recopie directe de la mémoire, on n'a pas à savoir comment est codé chaque nombre

• **Modes d'ouverture de fichiers binaires :**
- `"rb"` (read) : lecture
- `"wb"` (write) : écriture (le fichier est écrasé s'il existe)
- `"ab"` (append) : écriture à la fin d'un fichier existant

Scanned with CamScanner

Écriture-lecture de fichiers binaires

• **Ecriture** d'un **bloc** de données en binaire

int **fwrite**(void *source, int taille_type, int nombre, FILE *flot)

fwrite :
- **Ecrit** tout un bloc de données en un seul appel
- **Retourne** un entier = nombre d'éléments effectivement écrits

Exemple :

Pointeur sur les données

```
fwrite(tableau,sizeof(float),DIM,fichier);
```

Pointeur sur le fichier

• **Lecture** d'un **bloc** de données en binaire

int **fread**(void *destination, int taille_type, int nombre, FILE *flot)

fread :
- **Lit** un bloc de données en un seul appel
- **Retourne** un entier = nombre d'éléments effectivement lus

Exemple :

Pointeur sur les données **Pointeur sur le fichier**

```
fread(tableau,sizeof(float),DIM,fichier);
```

Activer Win

Scanned with CamScanner

Exemple: fichiers binaires

```
#define DIM 10000
void main()
{
    int     i;
    double  sum,tab1[DIM],tab2[DIM];
    FILE   *fichier;
    // Remplissage du tableau
    for(i=0;i<DIM;i++)
        tab1[i]=i*atan(1);
    // Ecriture du fichier au format binaire
    fichier = fopen("essai2.bin","wb");
    if (fichier != NULL)
    {
        fwrite(tab1,sizeof(double),DIM,fichier);
        fclose(fichier);
    }
    // Lecture du fichier
    fichier = fopen("essai2.bin","rb");
    if (fichier != NULL)
    {
        fread(tab2,sizeof(double),DIM,fichier);
        fclose(fichier);
    }
}
```

Scanned with CamScanner